MASSACHUSETTS INSTITUTE OF TECHNOLOGY
ARTIFICIAL INTELLIGENCE LABORATORY

A.I. T.R. No. 753 May 1985

KBEmacs: A Step Toward the Programmer's Apprentice

by

Richard C. Waters

ABSTRACT

The Knowledge-Based Editor in Emacs (KBEmacs) is the current demonstration system implemented as part of the Programmer's Apprentice project. KBEmacs is capable of acting as a semi-expert assistant to a person who is writing a program — taking over some parts of the programming task. Using KBEmacs, it is possible to construct a program by issuing a series of high level commands. This series of commands can be as much as an order of magnitude shorter than the program it describes.

KBEmacs is capable of operating on Ada and Lisp programs of realistic size and complexity. Although KBEmacs is neither fast enough nor robust enough to be considered a true prototype, both of these problems could be overcome if the system were to be reimplemented.

This report describes research done at the Artificial Intelligence Laboratory of the Massachusetts Institute of Technology. Support for the laboratory's artificial intelligence research has been provided in part by the Advanced Research Projects Agency of the Department of Defense under Office of Naval Research contract N00014-80-C-0505, in part by National Science Foundation grants MCS-7912179 and MCS-8117633, and in part by the IBM Corporation.

Dedicated To

Dr. John L. Weil

Acknowledgments

I would particularly like to acknowledge the assistance of my partner Charles Rich who has made important contributions to every aspect of my work for a decade.

KBEmacs is the result of a group effort which began with the original Programmer's Apprentice proposal of Charles Rich and Howard Shrobe. Over the years, many other people have contributed to that effort. Gerald Sussman has been an inspiration and mentor for us all. Kent Pitman implemented the user interface for KBEmacs and assisted with many other aspects of the system. David Cyphers developed the initial program documentation facilities. Daniel Brotsky, David Chapman, Roger Duffey, Gregory Faust, Daniel Shapiro, Peter Sterpe, and Linda Zelinka contributed ideas to KBEmacs while working on related parts of the Programmer's Apprentice project. Special thanks are due Roger Racine (of the C.S. Draper Laboratory) for his assistance with regard to Ada and Crisse Ciro for her help with the illustrations.

Contents

I - Introduction

The long term goal of the Programmer's Apprentice project is to develop a theory of programming (i.e., how expert programmers understand, design, implement, test, verify, modify, and document programs) and to automate the programming process. Recognizing that fully automatic programming is very far off, the current research is directed toward the intermediate goal of developing an intelligent computer assistant for programmers called the Programmer's Apprentice (PA). The intention is for the PA to act as a junior partner and critic, keeping track of details and assisting with the easy parts of the programming process while the programmer focuses on the hard parts of the process.

The Knowledge-Based Editor in Emacs (KBEmacs) is the current demonstration system implemented as part of the PA project. KBEmacs falls short of the PA because it focuses only on the task of program construction and because the depth of its understanding of a program is quite limited. However, KBEmacs demonstrates several of the capabilities of the PA and is a useful tool in its own right. The principal benefit of KBEmacs is that it makes it possible to construct a program rapidly and reliably by combining algorithmic fragments stored in a library.

Goals of the PA Project

The PA project is pursuing two goals in parallel. On the one hand, the project uses programming as a domain in which to investigate human problem solving behavior. On the other hand, the project seeks to improve programmer productivity by developing programming tools based on AI techniques.

Dramatic improvements in programmer productivity have been frustratingly hard to achieve. One reason for this is that programming is a complex task which consists of a variety of subtasks (e.g., requirements analysis, design, implementation, testing, and maintenance). Since each of these subtasks is a significant part of the process as a whole, it is impossible to get a dramatic increase in productivity without addressing at least most of these subtasks.

Looking back over the history of programming, only one development stands out as truly dramatic — the introduction of high level languages. High level languages have had a positive impact on almost every phase of the programming process by representing programs in a more concise and understandable way. They do this by delegating a variety of low level programming decisions (e.g. register allocation) to a compiler.

Using AI techniques, it may soon be possible to get a second dramatic improvement in programmer productivity by developing programming tools which can automatically perform middle level programming decisions (e.g., data structure selection). AI techniques make it possible to represent a great deal of knowledge about programming in general and then use this knowledge to understand particular programs. As has been demonstrated in other areas where AI techniques have been applied, this approach opens the door to intelligent behavior.

Although AI techniques hold considerable promise as the basis for advanced programming tools, a great deal more work has to be done. The PA project seeks both to develop the additional techniques which will be needed and to build demonstration systems which illustrate the potential for AI-based tools.

History of the Programmer's Apprentice Project

Figure 1 shows how KBEmacs fits into the PA project as a whole. The project consists of two principal lines of research: *foundations* and *demonstrations*. The first line seeks to develop new representation and reasoning techniques which can serve as the foundation for the PA. The second line of research seeks to construct demonstration systems based on these techniques and to experiment with how the PA might assist a programmer.

Besides laying out the basic concept of the PA, the most important contribution of the initial phase of the PA project (up to 1976) was the design of the *plan formalism* for representing knowledge about particular programs and about programming in general (see [Rich & Shrobe 76,78] and [Waters 76]). The plan formalism serves as the "mental language" of KBEmacs.

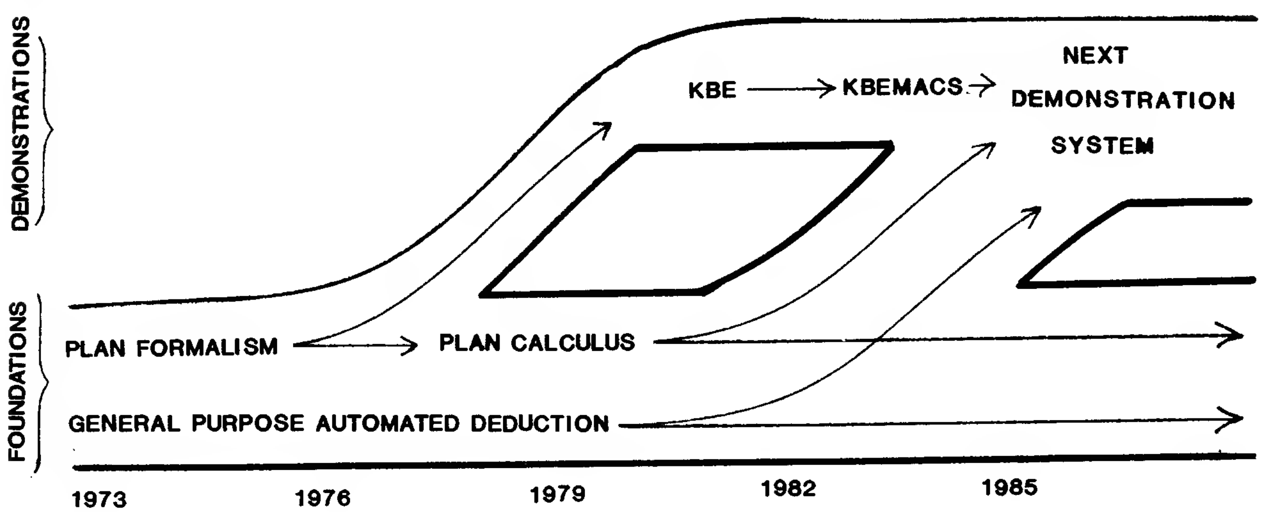

Figure 1: History of the Programmer's Apprentice project.

The first demonstration system constructed as part of the PA project was the Knowledge-Based Editor (KBE). KBE (see [Waters 82a]) was a program editor which made it possible to operate directly on the algorithmic structure of a program rather than on its textual or syntactic structure. Like KBEmacs, the power of KBE came principally from the ability to construct a program out of algorithmic fragments.

The topic of this report is KBEmacs, the second demonstration system. As the name Knowledge-Based Editor *in Emacs* is intended to imply, the most obvious difference between KBEmacs and KBE is that KBEmacs is tightly integrated with a standard Emacs-style [Stallman 81] program editor. The integration makes it possible for the programmer to freely intermix knowledge-based program editing with text-based and syntax-based program editing. As discussed in Chapter IV, KBEmacs extends KBE in a number of other ways, coming a step closer to the PA. In particular, it increases the range of algorithmic fragments which can be manipulated; it allows a programmer to define new fragments; it supports the language Ada in addition to Lisp; and it can assist in the construction of program documentation.

During the implementation of KBE and then KBEmacs, work has continued in parallel on the fundamental underpinnings of the PA. The central part of this work has been the refinement and extension of the plan formalism. This has resulted in the development of the *plan calculus* [Rich 80,81] which extends the plan formalism by providing a firmer semantic basis and by increasing the range of information about programs and programming which can be represented. Work has also proceeded on the development of general purpose automated deduction methods which are appropriate for use in the domain of plans (see [Shrobe 79] and [Rich 82,85]).

There has been a significant amount of interaction between the work on foundations and the work on demonstrations. Several of the improvements in plans have been incorporated into demonstration systems and experience with the demonstration systems has motivated several of the improvements in plans. However, as highlighted by Figure 1, the two lines of research have been largely separate. In particular, the magnitude of the implementation required has forced there to be a large delay between the time when new fundamental concepts are developed and the time when they are incorporated into a demonstration system. Implementation has already begun on a new demonstration system which will incorporate all of the fundamental ideas developed to date.

Outline of This Report

This report focuses almost exclusively on KBEmacs. Other parts of the PA project are mentioned only when they are essential for giving a full understanding of KBEmacs. As a result, this report is not intended to be a summary of the project as a whole.

The heart of this report is a pair of scenarios showing KBEmacs in action. Chapter II shows the system being used to construct Lisp programs. Chapter III shows the system being used to construct Ada programs. *The scenarios show the output of an actual running system.* However, as will be discussed below, there are several reasons why the current system is merely a demonstration system and cannot be considered to be a true prototype. Given this, the scenarios are perhaps best looked at as a set of requirements for aspects of the PA.

Chapter IV evaluates KBEmacs from three points of view: as a stand alone programming tool, as a step toward the PA, and in relation to other, similar programming tools. Chapter V explains how KBEmacs is implemented. Chapter VI discusses the future directions of the PA project. The remainder of this chapter presents a brief example of using KBEmacs, discusses the basic AI concepts behind the system, and then summarizes the capabilities of the system.

An Example of Using KBEmacs

In order to give a feeling for the program construction capabilities of KBEmacs, this section presents a condensed excerpt from the scenario in Chapter III. In that chapter a programmer uses KBEmacs to implement several Ada [Ada 83] programs in the domain of business data processing. It is assumed that there is a data base which contains information about various machines (referred to as *units*) sold by a company and about the repairs performed on each of these units. In the middle of Chapter III the programmer constructs a program called `UNIT_REPAIR_REPORT` which prints out a report of all of the repairs performed on a given unit. The following directions might be given to a human assistant who was asked to write this program.

```
Define a simple report program UNIT_REPAIR_REPORT.  Enumerate the chain of
repairs associated with a unit record, printing each one.  Query the user for
the key (UNIT_KEY) of the unit record to start from.  Print the title
("Report of Repairs on Unit " & UNIT_KEY).  Do not print a summary.
```

A key feature of these directions is that they refer to a significant amount of knowledge that the assistant is assumed to possess. First, they refer to a number of standard programming algorithms — i.e., "simple report", "enumerating the records in a chain", "querying the user for a key". Second, they assume that the assistant understands the structure of the data base of units and repairs. Another feature of the directions is that, given that the assistant has a precise understanding of the algorithms to be used and of the data base, little is left to the assistant's imagination. Essentially every detail of the algorithm is spelled out, including the exact Ada code to use when printing the title.

As discussed in Chapter III, the following set of commands can be used to construct the program `UNIT_REPAIR_REPORT` using KBEmacs. The Ada program which results from these commands is shown on the next page.

```
Define a simple_report procedure UNIT_REPAIR_REPORT.
Fill the enumerator with a chain_enumeration of UNITS and REPAIRS.
Fill the main_file_key with a query_user_for_key of UNITS.
Fill the title with ("Report of Repairs on Unit " & UNIT_KEY).
Remove the summary.
```

A key feature of these commands is that they refer to a number of standard algorithms known to KBEmacs — i.e., "simple_report", "chain_enumeration", and "query_user_for_key". In addition, they assume an understanding of the structure of the data base. Each of the "`Fill`" commands specifies how to fill in a part of the simple_report algorithm.

Without discussing either the commands or the program produced in any detail, two important observations can be made. First, the commands used are very similar to the hypothetical directions for a human assistant. Second, a set of 5 commands produces a 56 line program. (The program would be even longer if it did not make extensive use of data declarations and functions defined in the packages `FUNCTIONS` and `MAINTENANCE_FILES`.)

The KBEmacs commands and the hypothetical directions differ in grammatical form, but not in semantic content. This is not surprising in light of the fact that the hypothesized commands were in actuality created by restating the knowledge-based commands in more free flowing English.

The purpose of this translation was to demonstrate that although the KBEmacs commands may be syntactically awkward, they are not semantically awkward. The commands are neither redundant nor overly detailed. They specify only the basic design decisions which underly the program. There is no reason to believe that any automatic system (or for that matter a person) could be told how to implement the program `UNIT_REPAIR_REPORT` without being told at least most of the information in the commands shown.

```
with CALENDAR, FUNCTIONS, MAINTENANCE_FILES, TEXT_IO;
use CALENDAR, FUNCTIONS, MAINTENANCE_FILES, TEXT_IO;
procedure UNIT_REPAIR_REPORT is
   use DEFECT_IO, REPAIR_IO, UNIT_IO, INT_IO;
   CURRENT_DATE: constant STRING := FORMAT_DATE(CLOCK);
   DEFECT: DEFECT_TYPE;
   REPAIR: REPAIR_TYPE;
   REPAIR_INDEX: REPAIR_INDEX_TYPE;
   REPORT: TEXT_IO.FILE_TYPE;
   TITLE: STRING(1..33);
   UNIT: UNIT_TYPE;
   UNIT_KEY: UNIT_KEY_TYPE;
   procedure CLEAN_UP is
      begin
         SET_OUTPUT(STANDARD_OUTPUT);
         CLOSE(DEFECTS); CLOSE(REPAIRS); CLOSE(UNITS); CLOSE(REPORT);
      exception
         when STATUS_ERROR => return;
      end CLEAN_UP;
begin
   OPEN(DEFECTS, IN_FILE, DEFECTS_NAME); OPEN(REPAIRS, IN_FILE, REPAIRS_NAME);
   OPEN(UNITS, IN_FILE, UNITS_NAME); CREATE(REPORT, OUT_FILE, "report.txt");
   loop
      begin
         NEW_LINE; PUT("Enter UNIT Key: "); GET(UNIT_KEY);
         READ(UNITS, UNIT, UNIT_KEY);
         exit;
      exception
         when END_ERROR => PUT("Invalid UNIT Key"); NEW_LINE;
      end;
   end loop;
   TITLE := "Report of Repairs on Unit " & UNIT_KEY;
   SET_OUTPUT(REPORT);
   NEW_LINE(4); SET_COL(20); PUT(CURRENT_DATE);
   NEW_LINE(2); SET_COL(13); PUT(TITLE); NEW_LINE(60);
   READ(UNITS, UNIT, UNIT_KEY);
   REPAIR_INDEX := UNIT.REPAIR;
   while not NULL_INDEX(REPAIR_INDEX) loop
      READ(REPAIRS, REPAIR, REPAIR_INDEX);
      if LINE > 64 then
         NEW_PAGE; NEW_LINE; PUT("Page: "); PUT(INTEGER(PAGE-1), 3);
         SET_COL(13); PUT(TITLE); SET_COL(61); PUT(CURRENT_DATE); NEW_LINE(2);
         PUT("   Date     Defect   Description/Comment"); NEW_LINE(2);
      end if;
      READ(DEFECTS, DEFECT, REPAIR.DEFECT);
      PUT(FORMAT_DATE(REPAIR.DATE)); SET_COL(13); PUT(REPAIR.DEFECT);
      SET_COL(20); PUT(DEFECT.NAME); NEW_LINE;
      SET_COL(22); PUT(REPAIR.COMMENT); NEW_LINE;
      REPAIR_INDEX := REPAIR.NEXT;
   end loop;
   CLEAN_UP;
exception
   when DEVICE_ERROR | END_ERROR | NAME_ERROR | STATUS_ERROR =>
      CLEAN_UP; PUT("Data Base Inconsistent");
   when others => CLEAN_UP; raise;
end UNIT_REPAIR_REPORT;
```

The leverage that KBEmacs applies to the program construction task is illustrated by the order of magnitude difference between the size of the set of commands and the size of the program. A given programmer seems to be able to produce more or less a constant number of lines of code per day independent of the programming language being used. As a result, there is reason to believe that the order of magnitude size reduction provided by the KBEmacs commands would translate into an order of magnitude reduction in the time required to construct the program. It should be noted that since program construction is only a small part (around 10 percent) of the programming life cycle, this does not translate into an order of magnitude savings in the life cycle as a whole.

Another important advantage of KBEmacs is that using cliches enhances the reliability of the programs produced. Since cliches are intended to be used many times, it is economically justifiable to lavish a great deal of time on them in order to ensure that they are general purpose and bug free. This reliability is inherited by the programs which use the cliches. When using an ordinary program editor, programmers typically make two kinds of errors: picking the wrong algorithms to use and incorrectly instantiating these algorithms (i.e., combining the algorithms together and rendering them as appropriate program code). KBEmacs eliminates the second kind of error.

Key AI Ideas Underlying KBEmacs

Three basic AI ideas — the assistant approach, cliches, and plans — underlie the PA project as a whole and KBEmacs in particular. These ideas define the approach taken and are the basis for the capabilities of the system. A fourth idea — general purpose automated deduction — is an important aspect of the project as a whole, but is not used by KBEmacs.

The Assistant Approach

When it is not possible to construct a fully automatic system for a task, it is nevertheless often possible to construct a system which can assist an expert in the task. In addition to being pragmatically useful, the assistant approach can lead to important insights into how to construct a fully automatic system.

Figure 2 shows a programmer and an assistant interacting with a programming environment. Though presumably less knowledgeable, the assistant interacts with the tools in the environment (e.g., editors, compilers, debuggers) in the same way as the programmer and is capable of helping the programmer do what needs to be done. It is assumed that the programmer will not be able to delegate all of the work which needs to be done to the assistant and therefore will have to interact with the programming environment directly from time to time in order to do things which the assistant is not capable of doing.

The key issue in using an assistant effectively is *division of labor.* Since the programmer is more capable, the programmer will have to make the hard decisions about what should be done and what algorithms should be used. However, much of programming is quite mundane and can easily be done by an assistant. The key to cooperation between the programmer and the assistant is effective two-way communication — whose key in turn is *shared knowledge.* It would be impossibly tedious for the programmer to explain each decision to the assistant from first principles. Rather, the programmer needs to be able to rely on a body of intermediate-level shared knowledge in order to communicate decisions easily.

The discussion in the last two paragraphs applies equally well to human assistants and automated assistants. KBEmacs is intended to interact with a programmer in the same way that a human assistant might. The long range goal of the PA is to create a "chief programmer team" wherein the programmer is the chief and the PA is the team.

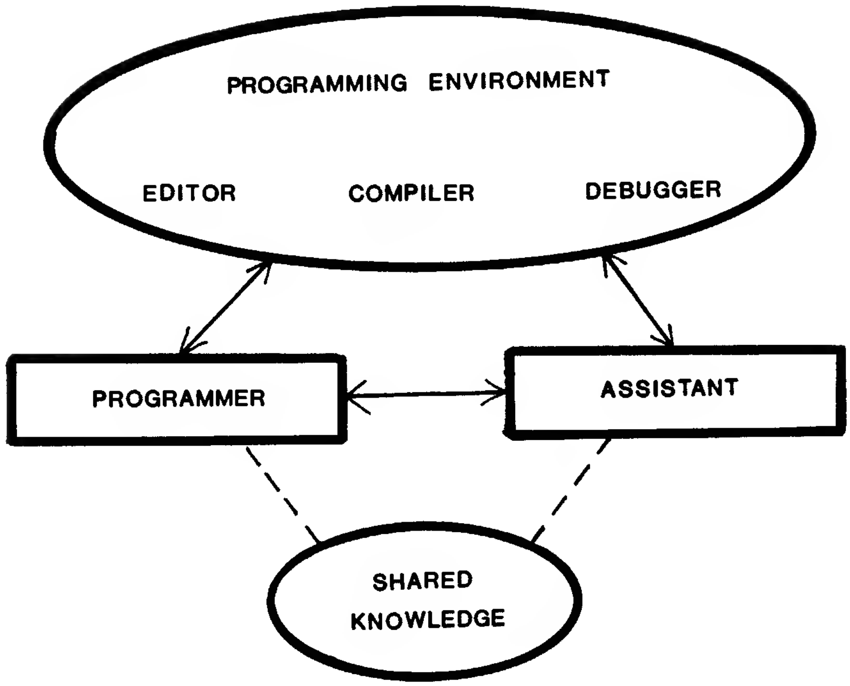

Figure 2: A programming assistant.

An important benefit of the assistant approach is that it is non-intrusive in nature. The assistant is available for the programmer to use, but the programmer is not forced to use it. Note that this contrasts sharply, for example, with program generators which completely take over large parts of the programming task and do not allow the programmer to have any control over them. A key goal of KBEmacs is to provide assistance to the programmer without preventing the programmer from doing simple things in the ordinary way. The intent is for the programmer to use standard programming tools whenever that makes things easy and to use KBEmacs only when doing so delivers real benefits.

A key part of the assistant approach as described above is the assumption that the assistant is significantly less knowledgeable than the programmer. There are situations where one might want an assistant system which was more knowledgeable than the programmer (e.g., a system which assists end users or neophyte programmers). However, KBEmacs does not attack these kinds of problems. The goal of KBEmacs is to make expert programmers super-productive rather than to make bad programmers good.

Cliches

The term *cliche* is used in this report to refer to a standard method for dealing with a task — a lemma or partial solution. In normal usage, the word cliche has a pejorative sound which connotes overuse and lack of creativity. However, it is not practical to be creative all of the time. For example, when implementing a program, it is usually better to construct a reasonable program rapidly, than to construct a perfect program slowly.

A cliche consists of a set of *roles* embedded in an underlying *matrix*. The roles represent parts of the cliche which vary from one use of the cliche to the next but which have well defined purposes. The matrix specifies how the roles interact in order to achieve the goal of the cliche as a whole.

As an example of a cliche in the domain of programming, consider searching a one dimensional structure. One way to do this is to use the cliche *sequential-search*. This cliche enumerates the elements of a structure one at a time, tests each element to see if it satisfies the goal of the search, and returns the first element which passes the test. If no element passes the test, then a special value signifying the failure of the search is returned.

The cliche sequential-search has three roles: the structure to be searched, the enumerator to use when searching the structure, and the test which defines the goal of the search. The matrix of the cliche specifies several different kinds of information. First, it specifies pieces of fixed computation which do not vary form one use of the cliche to the next. A simple example of this is the special value signifying failure in the cliche sequential-search. Most cliches have fixed computation which is more complex than mere constants.

Second, the matrix specifies the control flow and data flow which connect the roles with each other and with the fixed computation. For example, data flow connects the output of the enumerator with the input of the test and control flow specifies that the first time the test succeeds, the search should be terminated. Third, the matrix specifies various constraints on the roles (e.g., the constraint that the enumerator must be compatible with the data type of the structure to be searched).

When a cliche is used, it is *instantiated* by filling in the roles with computations which are appropriate to the task at hand. This creates an instance of the cliche which is specialized to the task. For example, in order to use the cliche sequential-search to find the first negative element of a vector of integers, the vector would be used as the structure to be searched, the enumerator would be filled with computation which enumerates a vector, and the test would be filled with a test for negativity.

Given a particular domain, cliches provide a vocabulary of relevant intermediate and high level concepts. Having such a vocabulary is essential for effective reasoning and communication in the context of the domain. It is important to note that this is just as important for human thought as it is for machine-based thought.

Both men and machines are limited in the complexity of the lines of reasoning they can develop and understand. In order to deal with more complex lines of reasoning, intermediate level vocabulary must be introduced which summarizes parts of the line of reasoning. Once this intermediate vocabulary is fully understood, it can be used to express the full line of reasoning in a sufficiently straightforward way.

Men and machines are similarly limited in the complexity of the descriptions they can communicate. Just as it is in general never practical to reason about something from first principles, it is in general never practical to describe something in full detail from first principles. Effective communication depends on the shared knowledge of an appropriate vocabulary between speaker and hearer.

An essential part of the cliche concept is *reuse*. Once something has been thought out (or communicated) and given a name, then it can be reused as a component in future thinking (communication). There is an overhead in that something must be thought out very carefully in order for it to serve as a truly reusable component. However, if successful, this effort can be amortized over many instances of reuse.

A corollary of the cliche idea is that a library of cliches is often the most important part of an AI system. In KBEmacs, a large portion of the knowledge which is shared between man and machine is in the form of a library of algorithmic cliches. This library can be viewed as being a machine understandable definition of the vocabulary programmers use when talking about programs.

Plans

Selecting an appropriate knowledge representation is the key to applying AI to any task. As a practical matter, the only way to perform a complex (as opposed to merely large) operation is to find a knowledge representation in which the operation can be performed in a relatively straightforward way. To this end, many AI systems make use of the idea of a *plan* — a representation which is abstract in that it deliberately ignores some aspects of a problem in order to make it easier to reason about the remaining aspects of the problem.

To be useful, a knowledge representation must express all of the information relevant to the problem at hand. The plan formalism used by KBEmacs is designed to represent two basic kinds of information: the structure of particular programs and knowledge about cliches. The structure of a program is expressed essentially as a flow chart where data flow as well as control flow is represented by explicit arcs. In order to represent cliches, added support is provided for representing roles and constraints.

Equally important, a knowledge representation must facilitate the operations to be performed. The two key operations performed by KBEmacs are simple reasoning about programs (e.g., determining the source of a data flow) and combining cliches together to create programs. The plan formalism is specifically designed to support these operations. For example, the fact that data flow is expressed by explicit arcs makes it easy to determine the source of a given data flow.

The Plan Formalism

The following briefly summarizes the plan formalism used by KBEmacs. Chapter V discusses the formalism in more detail. A plan is like a hierarchical flow chart wherein both the control flow and data flow are represented by explicit arcs. Figure 3 shows a diagram of a simple example plan — the plan for the cliche absolute-value.

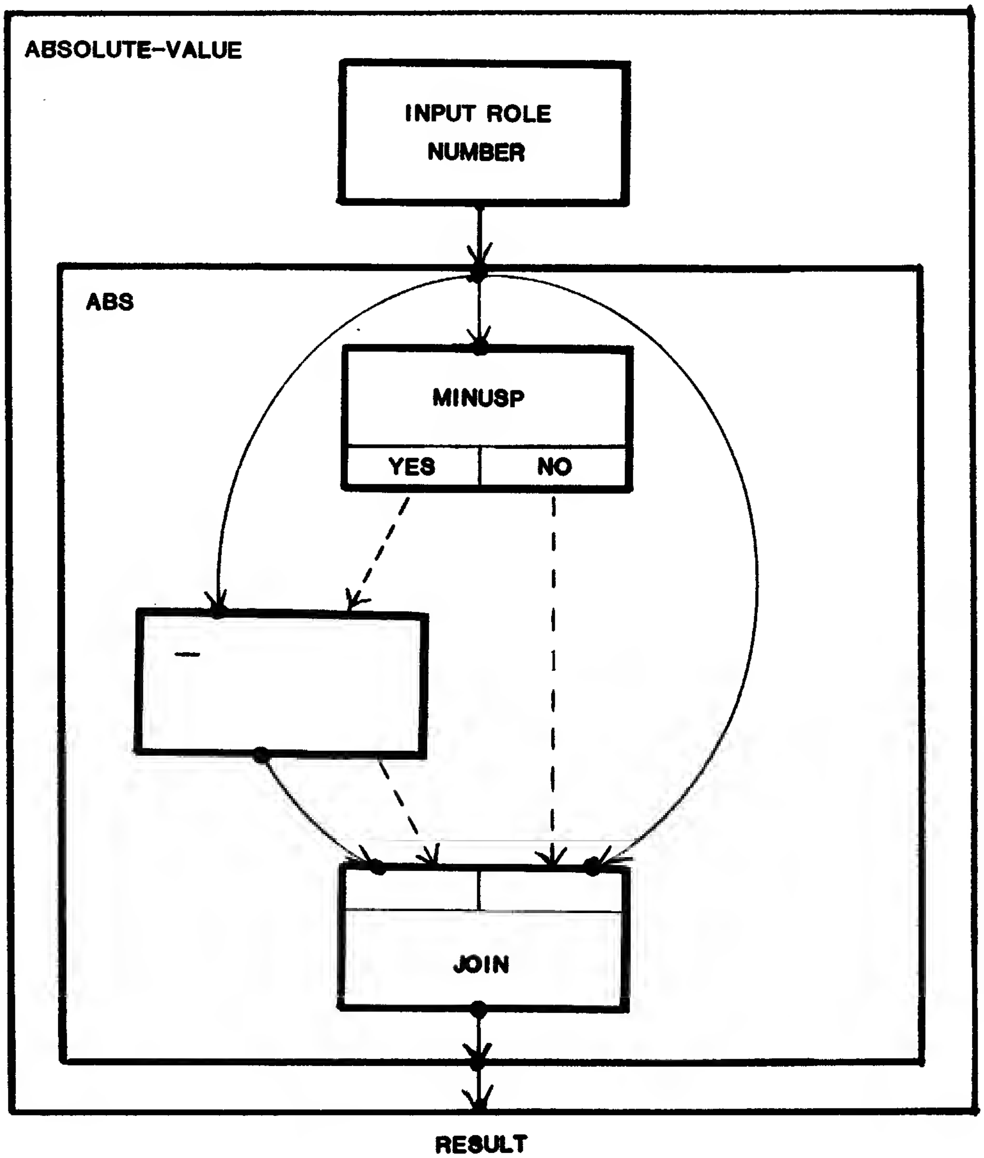

Figure 3: A plan for the cliche absolute-value.

The basic unit of a plan is a *segment* (drawn as a box in a plan diagram). A segment corresponds to a unit of computation. It has a number of *input ports* and *output ports* which specify the input values it receives and the output values it produces. It has a name which indicates the operation performed. A segment can either correspond to a primitive computation (e.g. the segment "-") or contain a subplan which describes the computation performed by the segment (e.g. the segment ABS). All of the computation corresponding to a single program or cliche is grouped together into one outermost segment. The roles of a cliche are represented as specially marked segments (e.g., the segment NUMBER).

As in a flow chart, control flow from one segment to another is represented by an explicit arc from the first segment to the second (drawn as a dashed arrow). Similarly, data flow is represented by an explicit arc from the appropriate output port of the source segment to the appropriate input port of the destination segment (drawn as a solid arrow). It should be noted that like a data flow diagram, and unlike an ordinary flowchart, data flow is the dominant concept in a plan. Control flow arcs are only used where they are absolutely necessary. In Figure 3, control flow arcs are necessary in order to specify that the operation "-" is performed only when the input number is less than zero.

A key feature of the plan formalism is that it abstracts away from the syntactic features of programming languages and represents the semantic features of a program directly. Whenever possible, it eliminates features which stem from the way things must be expressed in a particular programming language, keeping only those features which are essential to the actual algorithm. For example, a plan does not represent data flow in terms of the way it could be implemented in any particular programming language — e.g. with variables, or nesting of expressions, or parameter passing. Rather, it just records what the net data flow is. Similarly, no information is represented about how control flow is implemented.

Abstracting away from the syntactic features of a program has several advantages. One advantage is that it makes the internal operations of KBEmacs substantially programming language independent. Another advantage is that plans are much more canonical than program text. Programs (even in different languages) which differ only in the way their data flow and control flow is implemented correspond to the same plan.

A second important feature of the plan formalism is that it tries to make information as local as possible. For example, each data flow arc represents a specific communication of data from one place to another and, by the definition of what a data flow arc is, the other data flow arcs in the plan cannot have any effect on this. The same is true for control flow arcs. This locality makes it possible to determine what the data flow or control flow is in a particular situation by simply querying a small local portion of the plan.

The key benefit of the locality of data flow and control flow is that it gives plans the property of *additivity*. It is always permissible to put two plans side by side without their being any interference between them. This makes it easy for KBEmacs to create a program by combining the plans for cliches. All KBEmacs has to do is merely paste the pieces together. It does not have to worry about issues like variable name conflicts, because there are no variables.

A third important feature of plans is that the intermediate segmentation breaks a plan up into regions which can be manipulated separately. In order to ensure this separability, the plan formalism is designed so that nothing outside of a segment can depend on anything inside of that segment. For example, all of the data flow between segments outside of an intermediate segment and segments inside of an intermediate segment is channeled through input and output ports attached to the intermediate segment. As a result of this and other restrictions, when modifying the plan inside of a segment one can be secure in the knowledge that these changes cannot effect any of the plan outside of the segment.

Representing Loops

A final interesting aspect of plans is that loops are represented in a way which increases locality. Rather than representing loops by means of cycles in data flow and control flow, the plan is represented as a composition of computations applied to series of values (see [Waters 79]). For example, Figure 4 shows a plan for summing up the positive elements of a vector. It is composed of three subsegments. The first enumerates the elements in the input vector creating a series (or stream) of values. The second selects the positive elements of this series. The third sums up the selected values.

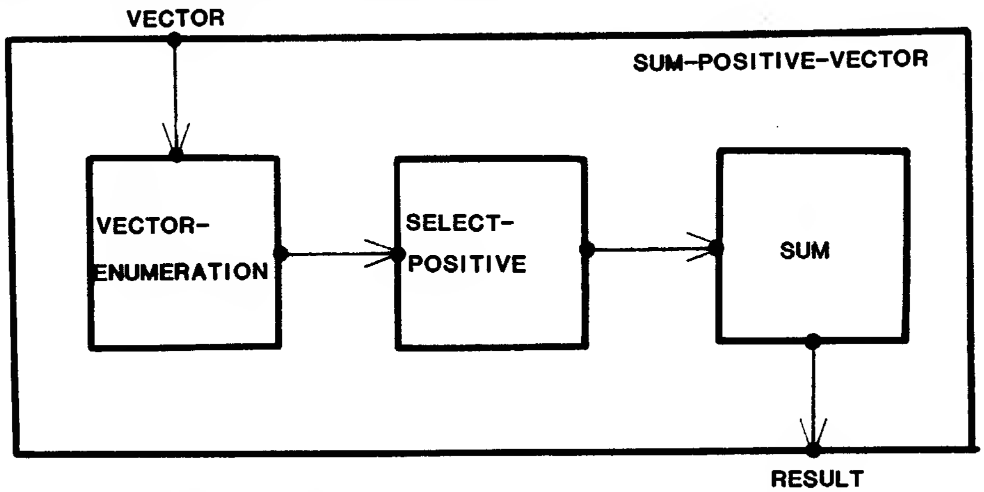

Figure 4: A plan for summing up the positive elements of a vector.

Representing a loop as a composition of computations on series has two important advantages. First, it increases locality. For example, it makes it possible to modify one of the computations without disturbing the others (e.g., the vector enumeration could be replaced by a list enumeration). Second, it highlights the similarity between related loops. For example, it makes explicit the fact that exactly the same summation cliche is used in a program which sums the positive elements of a vector as in a program which sums the lengths of a list of queues.

Representation Shift

One of the most powerful ideas underlying AI systems is the idea of a *representation shift* — shifting from a representation where a problem is easy to state but hard to solve to a representation which may be less obvious but in which the problem is easy to solve. Much of the power of KBEmacs is derived more or less directly from the representation shift from program text to the plan formalism.

From the point of view of a programmer using KBEmacs, Both the assistant approach and cliches are obvious features of the system. In contrast, plans are used only internally. KBEmacs goes to considerable lengths to make it appear to the user as if KBEmacs used program text (extended to support the notion of roles) as its only knowledge representation. In fact, Chapters II & III will make no mention of plans. The entire operation of KBEmacs will be presented from a purely textual point of view.

Overview of KBEmacs

As mentioned above, KBEmacs is the current demonstration system implemented as part of the PA project. It is implemented on the Symbolics Lisp Machine [Lisp 84] and is a restricted version of the PA in that it focuses primarily on the task of program construction. However, it illustrates a number of the capabilities which the PA is expected to possess.

Architecture

Figure 5 shows the architecture of the KBEmacs system. As discussed in Chapter V, KBEmacs maintains two representations for the program being worked on: program text and a plan. At any moment, the programmer can either modify the text or the plan. If the text is modified, then the analyzer module is used to create a new plan. If the plan is modified, the coder module is used to create new program text.

To modify the program text, the programmer can use the standard Emacs-style Lisp Machine program editor. This editor supports both text-based and syntax-based program editing. To modify the plan, the programmer can use the knowledge-based editor implemented as part of KBEmacs. This editor supports a set of knowledge-based commands which are phrased in a simple pseudo-English command language.

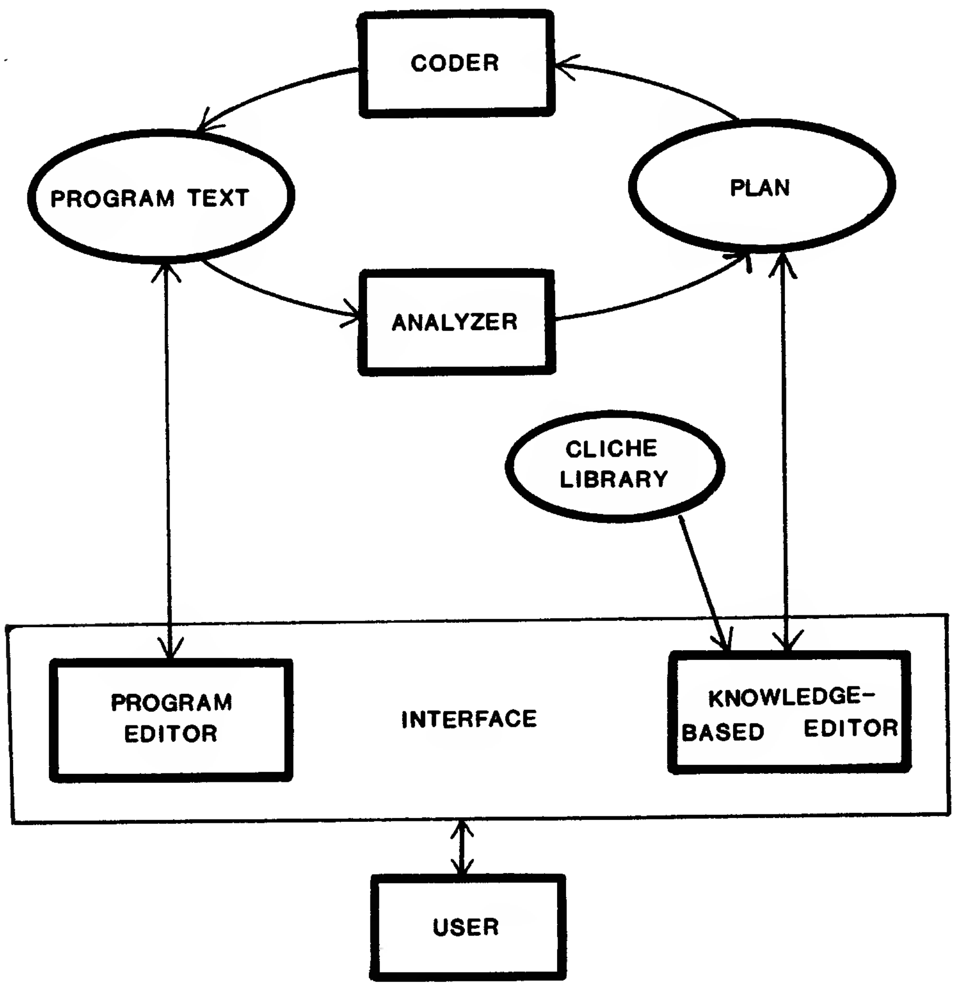

Figure 5: Architecture of KBEmacs.

An interface unifies ordinary program editing and knowledge-based editing so that they can both be conveniently accessed through the standard Lisp Machine editor. The knowledge-based commands are

supported as an extension of the standard editor command set and the results of these commands are communicated to the programmer by altering the program text in the editor buffer. The effect is the same as if a human assistant were sitting at the editor modifying the text under the direction of the programmer.

The major value of KBEmacs stems from the fact that it has a knowledge base of algorithmic cliches (the cliche library) and a significant amount of knowledge (procedurally embedded in the knowledge-based editor) about how to combine them. As seen in the example at the beginning of this chapter, a user can build up a program rapidly and reliably by selecting various algorithms to use and delegating to the system the task of combining them together to construct a program. However, the system is non-invasive because the user can fall back on ordinary program editing at any time.

As will be seen in the scenarios in the next two chapters, KBEmacs is a real running system which can operate on programs of realistic size and complexity. However, KBEmacs has three pragmatic limitations which render it usable only as a demonstration system rather than as a true prototype. First, it is too slow. Knowledge-based commands can take as long as five minutes to execute. In order for KBEmacs to be truly convenient to use, this time would have to be reduced to only a few seconds. Second, the system has evolved over a number of years in a very untidy fashion and is full of bugs. Pilot studies have indicated that both of these problems could be fixed by rewriting the system, but this would not be a simple task. Third, KBEmacs has an understanding of only a few dozen cliches. These are sufficient to support the scenarios shown in this report. However, a practical tool would have to have an understanding of hundreds if not thousands of cliches. Again, there is no fundamental reason why the appropriate cliches could not be entered into the system, but it would be a significantly difficult task.

Capabilities

As will be discussed at length in Chapters II & III, KBEmacs has many capabilities only one of which (rapid program construction in terms of cliches) is illustrated above. As discussed in Chapter IV, these capabilities can be divided into two classes: *strongly demonstrated* and *weakly demonstrated*. The following capabilities are strongly demonstrated in that they are supported in a general way, and little, if any, additional research would be needed before a full scale prototype system exhibiting these capabilities could be built.

Rapid program construction in terms of cliches — The programmer can construct programs quickly and reliably by combining cliches from KBEmacs' library of cliches.

User definition of cliches — The programmer can define new cliches as easily as he can define new subroutines.

Editing in terms of algorithmic structure — The programmer can operate on a program by using knowledge-based commands which refer to the logical structure of the algorithms being used.

Escape to the surrounding environment — While using KBEmacs the programmer is not prevented from using any of the standard programming tools. In particular, he can freely intermix text editing and knowledge-based editing.

Substantial programming language independence — The scenarios below show KBEmacs being used to operate on both Lisp and Ada programs.

The following capabilities are only weakly demonstrated. The weakness comes from the fact that the capabilities are not supported in a general way. Rather, only certain restricted aspects of each capability are supported in order to show what the capability would be like and to demonstrate that the PA approach has leverage on the capability.

A library of cliches — KBEmacs contains a very simple library of several dozen Lisp and Ada cliches.

Reasoning by means of constraints — KBEmacs uses simple constraint propagation in order to determine some of the consequences of the programmer's design decisions.

Taking care of details — KBEmacs is able to automatically take care of some kinds of programming details. For example, it can generate most of the variable declarations in an Ada program.

Program Modification — KBEmacs supports several commands which make it easier to modify a program. For example, it makes it easy to change the way the role of a cliche is filled in.

Program Documentation — KBEmacs is able to create a simple comment describing a program.

Future Directions

As discussed in detail in Chapter VI, work in the PA project is proceeding in several directions. The largest amount of effort is going into the construction of the next demonstration system. Externally, this system will appear to the user to be very similar to KBEmacs. It will use the same interface and support the same basic capabilities. However, internally it will be very different form KBEmacs. It will be based on the plan calculus [Rich 80,81] as opposed to the simpler plan formalism used by KBEmacs. In addition, the reasoning performed by the new system will be supported by a general purpose reasoning system [Rich 82,85] tailored to support the plan calculus, rather than by special purpose procedures. A primary goal of the new system will be to strongly demonstrate the capabilities which are only weakly demonstrated by KBEmacs. Another goal is to extend the capabilities of the system into the area of bug detection and automatic algorithm selection.

Another area of activity centers around various applications of the ideas behind KBEmacs. A particularly interesting application is the Tempest editor implemented by Sterpe [Sterpe 85]. This editor is inspired by KBEmacs as a whole. It has a user extendable library of cliches (called templates) and makes it easy to combine these templates together. The key difference between Tempest and KBEmacs is that Tempest is purely text based. This greatly reduces the power of the system. However, it vastly simplifies the system and reduces the amount of computation which has to be performed by three or four orders of magnitude — Tempest runs with acceptable speed on an IBM PC.

Another application is the Lisp macro described in [Waters 83a]. This macro is inspired by the way loops are represented in the plan formalism. The macro makes it possible to represent computations as compositions of functions on series of values and uses some of the same algorithms which are used by the coder module of KBEmacs in order to compile the computations into efficient iterative loops.

A final aim of the PA project is to expand the scope of the research beyond program implementation. Initial work has begun on the design of a Requirements Analyst's Apprentice. The intention is to focus on the opposite end of the programming process from program implementation and apply the basic ideas of the assistant approach, cliches, plans, and general purpose automated deduction to the task of acquiring and modifying requirements.

Once a system has been constructed which captures requirements in a machine understandable form it should be possible to construct a system which bridges the gap between requirements analysis and implementation by supporting the process of program design. Eventually it should be possible to build a true Programmer's Apprentice which can assist in all phases of the programming process.

II - Lisp Scenario

This chapter illustrates the various ways KBEmacs can be used by presenting a scenario showing the construction of several Lisp programs. The chapter is divided into five sections. The first section discusses Lisp programming cliches. (Appendix A presents the complete Lisp Cliche library.) The second section shows KBEmacs being used to construct a report program through the top-down combination of cliches. The third section shows KBEmacs being used to construct a numerical program through the bottom-up combination of cliches. The fourth section shows KBEmacs being used to assist in program modification and documentation. The fifth section shows the definition of a new cliche. It also illustrates a third method of program construction — the lateral modification of almost-right cliches.

Lisp Cliches

As discussed in Chapter I, the heart of KBEmacs is the library of programming cliches. These cliches provide the basic vocabulary used in the communication between man and machine and embody most of the knowledge which is shared between the programmer and KBEmacs. One of the basic assumptions underlying KBEmacs is that the knowledge of cliches is indeed shared between man and machine — i.e., that the programmer is aware of at least the basic features of the various cliches. In consonance with this, it is important for the reader to develop an understanding of what these cliches are like before looking at the scenario below.

A Simple Cliche

As a simple example of a cliche, consider the following definition of how to square a number. Lisp cliches are defined by using the form `DEFINE-CLICHE`. This form specifies the name of the cliche, some declarations describing the cliche, and the computation corresponding to the cliche. The roles of a cliche are represented by `{...}`. Here, the only role is the number to be squared. (The Lisp function `(^ x y)` computes x^y.)

```
(DEFINE-CLICHE SQUARING
    (PRIMARY-ROLES (NUMBER)
     DESCRIBED-ROLES (NUMBER)
     COMMENT "computes the square of {the number}")
  (^ {the input number} 2))
```

When communicating with KBEmacs, an instance of a cliche is referred to by using an indefinite noun phrase — e.g., "`a squaring of X`". Such a phrase specifies the name of the cliche, and may specify values which fill roles of the cliche. The `PRIMARY-ROLES` declaration specifies which roles can be specified this way, and the order in which they must be specified. (The language for communicating with KBEmacs is described in detail in Chapter V.)

The `COMMENT` declaration in the cliche definition is used for generating brief descriptions of instances of the cliche. Role references (represented by `{...}`) in the `COMMENT` string are replaced by descriptions of the actual computations filling the roles. For example, the cliche instance referred to in the last paragraph would be briefly described by saying that it "`computes the square of X`".

The `DESCRIBED-ROLES` declaration is used for generating in-depth descriptions of instances of the cliche. As will be discussed in detail in conjunction with the scenario below, this declaration specifies which roles should be described in detail when constructing a comment for a program.

Machine Understandable Program Annotation

Before looking at additional cliches it is useful to look at the notation `{...}` in more depth. The most common use of `{...}` is to represent roles, however, it is not the only use. As discussed in detail in Chapter V, this extension to Lisp syntax is provided to represent a variety of machine understandable program annotation.

There are two basic forms of the notation `{...}`. The first form is {*code*, *annotation*} which describes some feature of an existing part of a program. The second form is {*annotation*} which describes a non-existent part of the program. In the latter case, the form acts as a place holder that specifies how the non-existent part fits into the computation as a whole.

If the annotation in a `{...}` form begins with the word "`the`" then it specifies the name of a role (e.g., `{the input number}`). (Lower case is used when printing out the annotation in a `{...}` form in order to differentiate the annotation from the surrounding code.) If code is provided as part of the annotation (e.g., `{(CAR LIST), the output element}`) then this indicates that the role is already filled in.

If a role name contains the word "`input`" then it is treated as logically being an input to the containing cliche. Similarly, if a role name contains the word "`output`" then it is treated as logically being an output of the containing cliche. Information about logical inputs and outputs is used to help determine the data flow between cliches when they are combined together. Most roles are neither inputs nor outputs but rather correspond to some internal part of the computation.

The name of a role can either be a simple noun phrase (e.g., `{the input number}`) or a compound phrase (e.g., `{the empty-test of the enumerator}`). As will be discussed below, a compound phrase implicitly specifies a higher level compound role (e.g., the enumerator) which contains the indicated sub-role.

When `{...}` annotation acts as a place holder for a role, it can either appear in the form of a simple quantity (e.g., `{the input number}`) or in the form of a function call (e.g., `({the operation} DATA)`). If the role annotation is in the form of a function call, then it specifies the arguments which are expected to be used by the computation filling the role. In the example above, the annotation specifies that the function filling the operation role should use as an input the value of the variable `DATA`.

A kind of annotation unrelated to roles is illustrated by the form `{DATA, modified}`. This specifies that the given instance of the variable `DATA` is being side-effected. For example, in the form `(F X {Y, modified})` the annotation specifies that the function `F` side-effects its second argument. This kind of annotation is used by KBEmacs as part of the basis for understanding the side-effects in a program.

The Cliche List-Enumeration

The cliche squaring is very simple in nature, and could have been represented as a simple subroutine. A key feature of cliches is that they are capable of representing algorithmic fragments which cannot be expressed as simple subroutines. For example, the cliche list-enumeration (shown below) captures the concept of enumerating the elements of a list. This cliche can be combined together with other looping cliches in order to efficiently perform various operations on lists.

```
(DEFINE-CLICHE LIST-ENUMERATION
    (PRIMARY-ROLES (LIST)
     DESCRIBED-ROLES (LIST)
     COMMENT "enumerates the elements of {the list}")
  (LET* ((LIST {the input list}))
    (LOOP DO
      (IF ({NULL, the empty-test} LIST) (RETURN))
      {({CAR, the element-accessor} LIST), the output element}
      (SETQ LIST ({CDR, the step} LIST)))))
```

The cliche list-enumeration is a member of a general class of cliches referred to as *enumerators*, all of which have essentially the same roles. The input role (here the list) is the aggregate structure which is to be enumerated. The *empty-test* (here `NULL`) tests to see whether all of the elements in the aggregate structure have been enumerated and therefore whether the enumeration should be terminated. The *element-accessor* (here `CAR`) accesses the individual elements of the aggregate structure. The *step* (here `CDR`) steps from one element of the aggregate structure to the next. The output of the element-accessor is given a name (here the element) so that it can be conveniently referred to when communicating with KBEmacs.

It is interesting to note that the cliche squaring also has an output — the result of the squaring operation. However, this output is not given a special role name. Rather, it is assumed that like functions, must cliches will have a return value. The return value of a cliche is automatically treated as a kind of output role by KBEmacs. Explicit output roles are only used in situations (such as the one in the cliche list-enumeration) where the ordinary return value mechanisms are not sufficient to specify an output.

The Cliche Simple-Report

A cliche of central importance to the scenario is the cliche simple-report (shown below). This cliche specifies the high level structure of a simple report program. The cliche is significantly more complex than the previous ones in several ways. To start with, although it is written using standard Lisp Machine functions, it makes use of a number of functions which may be unfamiliar to users of other Lisp dialects.

The form `WITH-OPEN-FILE` combines the actions of opening a file and guaranteeing that the file will be closed even if an error or other interrupt occurs during the evaluation of the form. The opened file is bound to a variable (here `REPORT`) during the execution of the body of the form. The form `LET*` is the same as the standard Lisp form `LET` except that the bound-variable value pairs are processed sequentially instead of in parallel. The expression `(TIME:PRINT-CURRENT-TIME NIL)` returns a string of the form `"mm/dd/yy hh:mm:ss"` specifying the current date and time. The function `FORMAT` prints out textual information. Like the Fortran construct it is named after, it is inscrutable but convenient for specifying how output is to be formatted on the page. The form `(LOOP DO ...)` repetitively executes the forms in its body until one of these forms terminates the loop by evaluating `(RETURN)`. The form `(WHEN` *predicate* `...)` evaluates the forms in its body if and only if the *predicate* evaluates to non-`NIL`.

```
(DEFINE-CLICHE SIMPLE-REPORT
    (PRIMARY-ROLES (ENUMERATOR PRINT-ITEM SUMMARY)
     DESCRIBED-ROLES (FILE-NAME TITLE ENUMERATOR
                      COLUMN-HEADINGS PRINT-ITEM SUMMARY)
     COMMENT "prints a report of {the input structure of the enumerator}"
     CONSTRAINTS
      ((DEFAULT {the file-name} "report.txt")
       (DERIVED {the line-limit}
                (- 65
                   (SIZE-IN-LINES {the print-item})
                   (SIZE-IN-LINES {the summary})))))
  (WITH-OPEN-FILE (REPORT {the file-name} ':OUT)
    (LET* ((DATE (TIME:PRINT-CURRENT-TIME NIL))
           (LINE 66)
           (PAGE 0)
           (TITLE {the title})
           (DATA {the input structure of the enumerator}))
      (FORMAT REPORT "~5%~66: <~A~>~2%~66: <~A~>~%" TITLE DATE)
      (LOOP DO
        (IF ({the empty-test of the enumerator} DATA) (RETURN))
        (WHEN (> LINE {the line-limit})
          (SETQ PAGE (+ PAGE 1))
          (FORMAT REPORT "~/|~%Page:~3D~50: <~A~>~17A~2%" PAGE TITLE DATE)
          (SETQ LINE 3)
          ({the column-headings} {REPORT, modified} {LINE, modified}))
        ({the print-item} {REPORT, modified}
                          {LINE, modified}
                          ({the element-accessor of the enumerator} DATA))
        (SETQ DATA ({the step of the enumerator} DATA)))
      ({the summary} {REPORT, modified}))))
```

Like any cliche, the cliche simple-report specifies some standard computation (e.g., the printing of the title page), some roles to be filled in (e.g., the title itself), and the data flow and control flow which combine them together. The cliche has seven roles. The *file-name* is the name of the file which will contain the report being produced. The *title* is printed on a title page and, along with the page number, at the top of each succeeding page of the report. The *enumerator* enumerates the elements of some aggregate data structure. The *print-item* is used to print out information about each of the enumerated elements. The *line-limit* is used to determine when a page break should be inserted in the report. The *column-headings* are printed at the top of each page

of the report in order to explain the output of the print-item. The *summary* prints out some summary information at the end of the report. Note that the print-item, column-headings, and summary are all computations which side-effect the report file by sending output to it.

The enumerator is a compound role which has the four sub-roles (the input structure, the empty-test, the element-accessor, and the step) typical of an enumerator. These sub-roles can be filled individually, or they can be filled together as a unit by using an enumeration cliche (such as the cliche list-enumeration) which specifies values for each of them.

A significant part of the complexity of the cliche simple-report stems from the fact that code is included for keeping track of the page number and the line number, and for determining when a page break should occur. (Unlike some other languages, Lisp Machine Lisp does not provide any automatic support for these operations.) Under the assumption that only 66 lines (numbered 0 through 65) can be printed on a page, the line number is initially set to 66 in order to force a page break immediately after the title page is printed. A page break is triggered whenever the line number is greater than the line-limit. When a page break is triggered, the page number is incremented, a new page is started with the appropriate page headings, and the line number is reinitialized. Note that the column-headings and the print-item are both expected to appropriately update the line number.

An important aspect of the cliche simple-report is that it specifies two constraints on the roles. These are specified as part of the declarations at the beginning of the cliche definition. The first constraint specifies that `"report.txt"` should be used as the default name for the file containing the report. This name will be used unless the programmer specifies some other name.

The second constraint specifies that the line limit should be derived as 65 minus the number of lines printed by the print-item, and the number of lines printed by the summary. (Constraint expressions are specified as a combination of ordinary lisp code and `{...}` annotation referring to roles.) The constraint guarantees that, whenever the line number is less than or equal to the line limit, there will be room for both the print-item and the summary to be printed on the current page. Because the line-limit role is derived by this constraint the programmer never has to fill it explicitly, and the role will automatically be updated if the print-item or summary are changed.

An important aspect of cliches in general is illustrated by the fact that the cliche simple-report contains the above computation and constraints concerning line numbers, page breaks, and page numbers. The fact that a standard scheme for dealing with pagination is included in the cliche improves the productivity of the programmers using the cliche because, in general, they no longer have to worry about it. Perhaps more importantly, it improves the capability and reliability of the programs produced using the cliche because the programs contain a fully general purpose scheme for dealing with pagination which is at least internally consistent.

The Cliche Print-Out

As a final example of a cliche, consider the cliche print-out (shown below) which prints out an item using the function FORMAT. The key aspect of the cliche print-out is that, in addition to printing the item, it properly increments the line number. The cliche has three roles: the item to be printed, a format-string which specifies how to print the item, and the size-in-lines which specifies how many print lines are used by the format-string. The role size-in-lines is automatically derived by a constraint. The function SIZE-IN-LINES is capable of analyzing a format string (or any other output computation) and determining the maximum number of lines which will be required.

```
(DEFINE-CLICHE PRINT-OUT
    (PRIMARY-ROLES (FORMAT-STRING ITEM)
     DESCRIBED-ROLES (FORMAT-STRING ITEM)
     COMMENT "prints out {the item}"
     CONSTRAINTS
      ((DEFAULT {the format-string} "~%~A")
       (DERIVED {the size-in-lines} (SIZE-IN-LINES {the format-string}))))
   (FORMAT REPORT {the format-string} {the input item})
   (SETQ LINE (+ LINE {the size-in-lines})))
```

Suites of Cliches

An important issue underlying cliches in general is that they are not designed in isolation. Rather, cliches are typically defined in tightly knit groups, or suites, which are intended to be used together. This is illustrated by the cliches simple-report and print-out. An essential aspect of these cliches is that they incorporate the same conventions for how to keep track of the line number. Another example of the interrelationships between cliches can be seen in the fact that the cliche list-enumeration is specifically designed to fit into the enumerator role of the cliche simple-report.

Top-Down Implementation

This section begins the presentation of a scenario illustrating the use of KBEmacs. In the first part of the scenario, the programmer constructs a program (REPORT-TIMINGS) which prints a report. Given a list of timings (e.g., of some experiment) the program prints out the timings followed by their mean and standard deviation. In order to implement this program, the programmer proceeds in a top-down fashion by first specifying the top level cliche to use and then filling in the roles of this cliche.

Directions for a Human Assistant

Suppose that an expert programmer were asked to write the program REPORT-TIMINGS and decided to delegate the task to an inexperienced programmer who was his assistant. In order to tell his assistant what to do, the expert programmer might give directions like the following.

```
Define a simple report program REPORT-TIMINGS with one parameter, a list of
timings.  Print the title "Report of Reaction Timings (in msec.)".  Print
each timing and then print a summary showing the mean and deviation of the
timings.  Do not print any column headings.
```

A key aspect of the directions above is that they assume a significant amount of shared knowledge between the expert and his assistant. In particular, they assume that the assistant understands the term "simple report". Presumably, understanding this term includes an understanding of how to print a title page, and how to print headings at the top of the subsequent pages of the report. At a more detailed level it presumably includes an understanding of how to determine when page breaks should be introduced. It is precisely this shared knowledge that the cliche simple-report is trying to capture. As we shall see, by using cliches such as simple-report, a programmer can tell KBEmacs how to produce the program REPORT-TIMINGS by giving directions at a similar level of detail to the directions shown above.

The Layout of Screen Images

The scenario below is illustrated with a sequence of Lisp Machine screen images. Most of these screens show the Emacs-style Lisp Machine editor in which KBEmacs is embedded. Each of these screens has two parts (see Screen 1). The first few lines show a set of commands. The box which makes up most of the screen shows the Emacs buffer which is the result of these commands. The line below this box shows the editor mode line and the screen number. The mode line is composed of the name of the system (i.e., "KBEmacs"), the editing mode (e.g., "(LISP)" or "(TEXT)"), and the name of the file being edited.

In the command area of a screen, bold face italics is used to indicate output typed by the system as opposed to what the programmer types. In the editor buffer area of a screen, bold face italics is used to indicate the changes in the buffer (in comparison with the previous screen) caused by the commands. It should be noted that highlighting changed portions of the editor buffer is unfortunately not currently supported by KBEmacs. As a step in this direction, whenever KBEmacs modifies the editor buffer, it positions the editing cursor (indicated by □ in the screens) at the beginning of the first significant change in the buffer.

Integration with Emacs

Before beginning to construct the program REPORT-TIMINGS the programmer uses the standard Emacs command c-X c-F (find file) in order to create a file which will contain the program (see Screen 1). KBEmacs is implemented so that it is tightly integrated with Emacs. KBEmacs adds a variety of new editor commands without interfering with (or rendering obsolete) any of the standard commands. As a result, the scenario presented here uses standard editor commands in order to perform standard operations. Although it is not assumed that the reader is familiar with these commands, in the interest of brevity, the standard commands used are not described in detail. The Lisp Machine documentation [Lisp 84] describes them more fully.

The Knowledge-Based Command "Define"

In analogy with the standard Emacs command m-X (extended command), KBEmacs supports a new command s-X (typed by holding down the Lisp Machine keyboard shift key SUPER and typing an X) which creates a special window wherein the programmer can type a command requesting KBEmacs to do something. Once a command is typed, it is executed by typing the Lisp Machine Key <end>. These commands (referred to as knowledge-based commands) are specified in an extremely simple pseudo-English command language (described in detail in Chapter V). Each knowledge-based command is a verb followed by one of more noun phrases. If a word is typed using capital letters (e.g., "TIMINGS" as opposed to "parameter") it is assumed to be a literal name or a code fragment and is interpreted without regard for any special meaning it may have in the command language. Automatic word completion in the s-X window is used to facilitate typing of the command language.

In order to begin the construction of the program REPORT-TIMINGS, The programmer uses the knowledge-based command "Define" to specify the name of the program and its parameter. KBEmacs communicates the results of knowledge-based commands to the programmer (and to the rest of the Lisp Machine system) by directly modifying the text in the editor buffer. In this case, the empty program definition (DEFUN REPORT-TIMINGS (TIMINGS)) is inserted into the buffer.

```
c-X c-F Find file <KBE.DEMO>TIMINGS.LISP
(New File)
s-X Define a program REPORT-TIMINGS with a parameter TIMINGS. <end>
```

```
(DEFUN REPORT-TIMINGS (TIMINGS))
```

```
KBEmacs (LISP) <KBE.DEMO>TIMINGS.LISP
```

Screen 1

The Knowledge-based Command "`Insert`"

In Screen 2, the programmer uses the knowledge-based command "`Insert`" in order to fill in the body of the program with an instance of the cliche simple-report. In general, the "`Insert`" command specifies that an instance of a cliche is to be inserted into a program at the place where the editing cursor is positioned.

The code produced in Screen 2 is an instantiation of the cliche simple-report. For the most part this code is exactly the same as the code in the definition of this cliche (see above). However, there are several key differences. For example, the file-name and line-limit roles have been filled in since they are specified by constraints. (The line-limit role has been given the value `63` under the default assumption, made by the function `SIZE-IN-LINES`, that the print-item and summary will each take up one output line.)

Suppressing Annotation

Another difference between the code in Screen 2 and the code in the definition of the cliche is that the `{...}` annotation marking the file-name and the line-limit has been suppressed. Once a role is filled in, the annotation indicating where it is located is no longer displayed. The only exception to this is output role annotation which is, in general, retained in order to alert the programmer to the name of the output role.

Suppressing the annotation of filled in roles is an aesthetic choice made by KBEmacs. It was decided that for the task of program construction, suppressing such annotation is helpful because it reduces visual clutter and highlights the roles which are not yet filled in. In other situations (e.g., documentation) it might be beneficial for KBEmacs to redisplay this annotation.

Remembering Annotation Information

Although KBEmacs does not display annotation for filled in roles, it remembers this information as part of its understanding of the program. In addition, KBEmacs remembers what cliches have been used to build up a program. As will be illustrated shortly, there are a number of ways in which KBEmacs can use both of these types of information to assist the programmer.

The Purpose of the Code Produced by KBEmacs

The above discussion brings up the interesting question of exactly what is the purpose of the code produced by KBEmacs. Clearly it is not a complete representation of everything that KBEmacs knows about a program. Rather, it fulfills two quite distinct purposes.

The first purpose of the code is to serve as the medium of communication between KBEmacs and the rest of the Lisp Machine system. For example, KBEmacs must create ordinary Lisp code for a program so that the standard Lisp interpreter, debugger, and compiler can operate on the program. This purpose requires only that the code produced correctly represent the algorithms being used.

The second purpose of the code is to serve as the primary medium of communication between KBEmacs and the programmer. At each moment, the code summarizes the net effect of the knowledge-based commands given so far. Further, because the programmer can freely edit the code at any time, the code is a two way communication channel. In order to fulfill this purpose, the code must be easily readable. In the interest of readability, KBEmacs only displays a small part of the information it could display about a program.

```
s-X Insert a simple-report. <end>
```

```
(DEFUN REPORT-TIMINGS (TIMINGS)
  (WITH-OPEN-FILE (REPORT "report.txt" ':OUT)
    (LET* ((DATE (TIME:PRINT-CURRENT-TIME NIL))
           (LINE 66)
           (PAGE 0)
           (TITLE {the title})
           (DATA {the input structure of the enumerator}))
      (FORMAT REPORT "~5%~66: <~A~>~2%~66: <~A~>~%" TITLE DATE)
      (LOOP DO
        (IF ({the empty-test of the enumerator} DATA) (RETURN))
        (WHEN (> LINE 63)
          (SETQ PAGE (+ PAGE 1))
          (FORMAT REPORT "~|~%Page:~3D~50: <~A~>~17A~2%" PAGE TITLE DATE)
          (SETQ LINE 3)
          ({the column-headings} {REPORT, modified} {LINE, modified}))
        ({the print-item} {REPORT, modified}
                          {LINE, modified}
                          ({the element-accessor of the enumerator} DATA))
        (SETQ DATA ({the step of the enumerator} DATA)))
      ({the summary} {REPORT, modified}))))
```

The Commands `s-N` and `s-P`

In keeping with the general Emacs philosophy of providing a wide variety of convenient one character commands, KBEmacs provides a number of such commands (all of which utilize the shift key `SUPER`) which operate on programs. The command `s-N` moves to the first occurrence of role annotation after the current cursor position. (In Screen 3, the programmer uses the command `s-N` in order to move to the title role. Ordinary cursor motion commands could also have been used.) The command `s-P` moves to the first occurrence of role annotation prior to the current cursor position. The commands `s-N` and `s-P` are analogous to the standard Emacs commands `c-N` (move to the next line) and `c-P` (move to the previous line).

Direct Editing

In order to fill the title role, the programmer uses ordinary Emacs commands to type in the title string. (In the command area of the screen the phrase "*direct editing*" is used in lieu of an explicit listing of these commands since only their net effect is interesting.) The programmer could have used knowledge-based commands in order to fill in the title. However, he judged that ordinary editing would be simpler. An important goal of KBEmacs is to provide assistance to the programmer without preventing him from doing simple things in simple ways. The intent is for the programmer to use ordinary text editing whenever that makes things easy and to use knowledge-based commands only when they deliver real value — e.g., when instantiating the cliche simple-report.

s-N
direct editing

```
(DEFUN REPORT-TIMINGS (TIMINGS)
  (WITH-OPEN-FILE (REPORT "report.txt" ':OUT)
    (LET* ((DATE (TIME:PRINT-CURRENT-TIME NIL))
           (LINE 66)
           (PAGE 0)
           (TITLE "Report of Reaction Timings (in msec.)")
           (DATA {the input structure of the enumerator}))
      (FORMAT REPORT "~5%~66: <~A~>~2%~66: <~A~>~%" TITLE DATE)
      (LOOP DO
        (IF ({the empty-test of the enumerator} DATA) (RETURN))
        (WHEN (> LINE 63)
          (SETQ PAGE (+ PAGE 1))
          (FORMAT REPORT "~|~%Page:~3D~50: <~A~>~17A~2%" PAGE TITLE DATE)
          (SETQ LINE 3)
          ({the column-headings} {REPORT, modified} {LINE, modified}))
        ({the print-item} {REPORT, modified}
                          {LINE, modified}
                          ({the element-accessor of the enumerator} DATA))
        (SETQ DATA ({the step of the enumerator} DATA)))
      ({the summary} {REPORT, modified}))))
```

KBEmacs (LISP) <KBE.DEMO>TIMINGS.LISP

Screen 3

The Knowledge-based Command "Fill"

In Screen 4, the programmer specifies that the program should enumerate the list of timings. He does this by using the knowledge-based command "Fill" to fill in the enumerator of the program with an instance of the cliche list-enumeration. The fill command specifies that an instance of a cliche is to be used to fill in an unfilled role. Definite noun phrases in a knowledge-based command are disambiguated with respect to the position of the cursor. As a result, the phrase "the enumerator" in the knowledge-based command in Screen 4 is interpreted to mean the enumerator of the program REPORT-TIMINGS because the cursor was positioned in this program when the command was typed (see Screen 3).

Filling a Compound Role

The most important thing to notice about Screen 4 is that all four sub-roles which are part of the enumerator of the program have been filled in at once. The cliche list-enumeration specifies these four roles as a logical unit even though these roles are distributed through the program. (The cursor is positioned on the empty-test on the theory that changes to the body of a program are more interesting than changes to variable initializations.)

Another thing to notice about the program in Screen 4 is that it no longer uses the variable DATA but rather the more informative variable name LIST. The readability of a program depends to a surprising extent on how the data flow is implemented and, in particular, on what variable names are used. Therefore, KBEmacs works hard to use nesting of expressions where appropriate and to pick the best variable names possible. In this case, the variable name LIST is chosen because it is specifically suggested by the cliche list-enumeration.

An additional minor change in the program in Screen 4 is that the order of the bound-variable value pairs in the LET* has been changed. In general, KBEmacs puts the bound-variable value pairs in alphabetical order. There are only two exceptions to this. First, data flow may require some other order. Second, uninformative variable names such as DATA are put at the end of the list (see Screen 3.)

s-X Fill the enumerator with a list-enumeration of TIMINGS. <end>

```
(DEFUN REPORT-TIMINGS (TIMINGS)
  (WITH-OPEN-FILE (REPORT "report.txt" ':OUT)
    (LET* ((DATE (TIME:PRINT-CURRENT-TIME NIL))
           (LINE 66)
           (LIST TIMINGS)
           (PAGE 0)
           (TITLE "Report of Reaction Timings (in msec.)"))
      (FORMAT REPORT "~5%~66: <~A~>~2%~66: <~A~>~%" TITLE DATE)
      (LOOP DO
        (IF (NULL LIST) (RETURN))
        (WHEN (> LINE 63)
          (SETQ PAGE (+ PAGE 1))
          (FORMAT REPORT "~|~%Page:~3D~50: <~A~>~17A~2%" PAGE TITLE DATE)
          (SETQ LINE 3)
          ({the column-headings} {REPORT, modified} {LINE, modified}))
        ({the print-item} {REPORT, modified}
                          {LINE, modified}
                          {(CAR LIST), the output element})
        (SETQ LIST (CDR LIST)))
      ({the summary} {REPORT, modified}))))
```

The Knowledge-Based Command "Remove"

After specifying the enumerator, the programmer proceeds to fill in the rest of the program. Since no column headings are to be used in this program, the programmer uses the knowledge-based command "Remove" in order to get rid of the column-headings role (see Screen 5). Note that the editing cursor is positioned where the column-headings used to be.

Screen 5 is an example of the fact that in addition to filling in roles, the programmer can prune away unneeded ones. In general, cliches (such as simple-report) are defined so that they include a wide variety of features on the theory that it is easier for the programmer to prune things away than to think them up.

s-X Remove the column-headings. <end>

```
(DEFUN REPORT-TIMINGS (TIMINGS)
  (WITH-OPEN-FILE (REPORT "report.txt" ':OUT)
    (LET* ((DATE (TIME:PRINT-CURRENT-TIME NIL))
           (LINE 66)
           (LIST TIMINGS)
           (PAGE 0)
           (TITLE "Report of Reaction Timings (in msec.)"))
      (FORMAT REPORT "~5%~66:<~A~>~2%~66:<~A~>~%" TITLE DATE)
      (LOOP DO
        (IF (NULL LIST) (RETURN))
        (WHEN (> LINE 63)
          (SETQ PAGE (+ PAGE 1))
          (FORMAT REPORT "~|~%Page:~3D~50:<~A~>~17A~2%" PAGE TITLE DATE)
          (SETQ LINE 3))
        ({the print-item} {REPORT, modified}
                          {LINE, modified}
                          {(CAR LIST), the output element})
        (SETQ LIST (CDR LIST)))
      ({the summary} {REPORT, modified}))))
```

KBEmacs (LISP) <KBE.DEMO>TIMINGS.LISP

Screen 5

The Abbreviated Command s-F

In Screen 6, the programmer specifies how to print out each of the timings being enumerated. He does this by using the command s-F. In addition to the basic command s-X, KBEmacs supports a number of abbreviated commands which make it easier to type knowledge-based commands. Each of these commands creates the same interaction window as the command s-X. However, when the window is created it is initialized with a partial (or complete) knowledge-based command. The result of using one of the abbreviated commands is identical to using s-X and typing the knowledge-based command in full. In the screens in the scenario, initialized text is highlighted with bold face italics since it is supplied by the system.

The command s-F sets up a "Fill" command for the first unfilled role which textually follows the current cursor position (in this case the print-item). The programmer completes the command by specifying that the print-item role should be filled with the cliche print-out. The programmer specifies the format-string as a literal piece of code in the knowledge-based command. (The format string forces a new line, prints 5 spaces, and then prints out an 8 digit decimal number.) The fact that program text can be directly included in a knowledge-based command is convenient in many situations.

Deducing Data Flow Connections

The most difficult part of filling in a role with an instance of a cliche is deciding how to connect up the data flow. This is done by comparing the data flow environment of the unfilled role with the data flow requirements of the cliche.

The print-item role in Screen 5 (reproduced below) specifies that this role takes as inputs the output file REPORT, the line number variable LINE, and the output element of the enumerator. In addition, it specifies that REPORT and LINE are modified by the print-item and therefore are outputs of the role.

```
({the print-item} {REPORT, modified}
                  {LINE, modified}
                  {(CAR LIST), the output element})
```

The definition of the cliche print-out (reproduced below) specifies that the cliche has a logical input role called item. It also specifies that the cliche has two additional inputs REPORT and LINE because these values are referenced as free variables in the body of the cliche. In addition, these values are outputs of the cliche because they are modified by the body of the cliche. (The variable REPORT is side-effected by the function FORMAT and the variable LINE is assigned to.)

```
(DEFINE-CLICHE PRINT-OUT
     (PRIMARY-ROLES (FORMAT-STRING ITEM)
      DESCRIBED-ROLES (FORMAT-STRING ITEM)
      COMMENT "prints out {the item}"
      CONSTRAINTS
       ((DEFAULT {the format-string} "~%~A")
        (DERIVED {the size-in-lines} (SIZE-IN-LINES {the format-string}))))
    (FORMAT REPORT {the format-string} {the input item})
    (SETQ LINE (+ LINE {the size-in-lines})))
```

When the print-item role is filled with an instance of the cliche print-out the data flow is connected up as follows. The REPORT and LINE inputs to the print-out are connected to the REPORT and LINE arguments of the print-item role because they have the same names. The same identity of names connects the REPORT and LINE outputs of the print-out to the side-effected outputs of the print-item role. The input item role of the print-out is connected to the output element of the enumerator because this element is the only remaining argument of the print-item which has not been accounted for. The return value of the print-out is not connected to anything because the return value of the print-item role is not used for anything.

s-F ***Fill the print-item with*** a print-out of "~&~5X~8:D". <end>

```
(DEFUN REPORT-TIMINGS (TIMINGS)
  (WITH-OPEN-FILE (REPORT "report.txt" ':OUT)
    (LET* ((DATE (TIME:PRINT-CURRENT-TIME NIL))
           (LINE 66)
           (LIST TIMINGS)
           (PAGE 0)
           (TITLE "Report of Reaction Timings (in msec.)"))
      (FORMAT REPORT "~5%~66: <~A~>~2%~66: <~A~>~%" TITLE DATE)
      (LOOP DO
        (IF (NULL LIST) (RETURN))
        (WHEN (> LINE 63)
          (SETQ PAGE (+ PAGE 1))
          (FORMAT REPORT "~|~%Page:~3D~50: <~A~>~17A~2%" PAGE TITLE DATE)
          (SETQ LINE 3))
        (FORMAT REPORT "~&~5X~8:D" {(CAR LIST), the output element})
        (SETQ LINE (+ LINE 1))
        (SETQ LIST (CDR LIST)))
      ({the summary} {REPORT, modified}))))
```

KBEmacs (LISP) <KBE.DEMO>TIMINGS.LISP

Screen 6

Asking What Needs to be Done

Up to this point in the scenario, KBEmacs has been used solely to assist in program construction. Although this is the dominant mode of interaction with KBEmacs, there are many other ways in which the system can support the programmer. For example, KBEmacs keeps track of what remains to be done in order for a program to be complete. In Screen 7, the programmer asks the system to report what still needs to be done with the program `REPORT-TIMINGS`. (He uses the abbreviated command `s-W` to save typing.) KBEmacs reports that the summary role still needs to be filled. In addition to unfilled roles, the system will also list any output roles which have not been used for anything.

In order to fill the summary, the programmer moves the editing cursor to the position of the summary role, and uses ordinary editing commands to type in an expression which prints out a summary showing the mean and deviation of the timings. In this expression he assumes the existence of a subroutine `MEAN-AND-DEVIATION` which returns the the mean and deviation of a list of numbers.

```
s-W What needs to be done? <end>
for the function REPORT-TIMINGS
 Fill the summary.
s-N s-N direct editing
```

```
(DEFUN REPORT-TIMINGS (TIMINGS)
  (WITH-OPEN-FILE (REPORT "report.txt" ':OUT)
    (LET* ((DATE (TIME:PRINT-CURRENT-TIME NIL))
           (LINE 66)
           (LIST TIMINGS)
           (PAGE 0)
           (TITLE "Report of Reaction Timings (in msec.)"))
      (FORMAT REPORT "~5%~66:<~A~>~2%~66:<~A~>~%" TITLE DATE)
      (LOOP DO
        (IF (NULL LIST) (RETURN))
        (WHEN (> LINE 63)
          (SETQ PAGE (+ PAGE 1))
          (FORMAT REPORT "~|~%Page:~3D~50:<~A~>~17A~2%" PAGE TITLE DATE)
          (SETQ LINE 3))
        (FORMAT REPORT "~&~5X~8:D" {(CAR LIST), the output element})
        (SETQ LINE (+ LINE 1))
        (SETQ LIST (CDR LIST)))
      (FORMAT REPORT "~2&~{mean:~8:D  (deviation: ~:D)~}"
              (MEAN-AND-DEVIATION TIMINGS))))
```

The Knowledge-Based Command "Finish"

In Screen 8, the programmer uses the knowledge-based command "Finish" (triggered by the abbreviated command s-<end>) in order to signal that he is finished implementing the program REPORT-TIMINGS. This command first checks that the program is indeed completed. It would complain if there were any unfilled roles or any output roles which were not used for anything. Next, the command removes all output role annotation (in this case, for the role element). This is done in order to render the program in completely standard Lisp. This annotation can be reintroduced at a later time if the programmer desires.

Analyzing Direct Editing Done by the Programmer

Before executing any knowledge-based command (e.g., s-X, s-F, or as here s-<end>), KBEmacs first analyzes the effects of any direct editing which the programmer has performed. Here, KBEmacs concludes that the code added by the programmer in Screen 7 is intended to fill the summary role since the only change in the program is that the new code replaces the summary textually. (KBEmacs waits until the next knowledge-based command is used in order to avoid analyzing partially edited programs which are in inconsistent states.)

This same kind of analysis occurred after the programmer used direct editing to fill in the title role in Screen 3. However, it was not mentioned there because it did not lead to any interesting results. Here however, KBEmacs changes the line-limit role of the program from 63 to 62.

The line-limit is changed due to the constraint on this role which is part of the definition of the cliche simple-report. As mentioned above, the value 63 was originally chosen under the assumption that the print-item and the summary would each require one output line. The line-limit did not change when the programmer filled the print-item, because he filled it with a computation which did require one output line. However, by using the FORMAT code ~2& the programmer filled the summary with a computation which requires two output lines.

Enhanced Program Reliability

The automatic updating of the line-limit role is a good example of the way KBEmacs can enhance program reliability. The main leverage KBEmacs applies to the reliability problem is that each cliche is internally consistent. The use of constraints can help maintain this consistency.

It is interesting to note that if KBEmacs did not update the line-limit role, the programmer would probably not have realized that it needed to be updated. The bug which would result, though minor, would have the pernicious quality of being rather hard to detect during program testing since the bug only manifests itself when the program attempts to print the summary as the last line of a page.

s-<end> ***Finish editing the function REPORT-TIMINGS. <end>***

```
(DEFUN REPORT-TIMINGS (TIMINGS)
  (WITH-OPEN-FILE (REPORT "report.txt" ':OUT)
    (LET* ((DATE (TIME:PRINT-CURRENT-TIME NIL))
           (LINE 66)
           (LIST TIMINGS)
           (PAGE 0)
           (TITLE "Report of Reaction Timings (in msec.)"))
      (FORMAT REPORT "~5%~66:<~A~>~2%~66:<~A~>~%" TITLE DATE)
      (LOOP DO
        (IF (NULL LIST) (RETURN))
        (WHEN (> LINE 62)
          (SETQ PAGE (+ PAGE 1))
          (FORMAT REPORT "~|~%Page:~3D~50:<~A~>~17A~2%" PAGE TITLE DATE)
          (SETQ LINE 3))
        (FORMAT REPORT "~&~5X~8:D" (CAR LIST))
        (SETQ LINE (+ LINE 1))
        (SETQ LIST (CDR LIST)))
      (FORMAT REPORT "~2&~{mean:~8:D  (deviation: ~:D)~}"
              (MEAN-AND-DEVIATION TIMINGS)))))
```

```
KBEmacs (LISP) <KBE.DEMO>TIMINGS.LISP
```

Screen 8

Integration With the Lisp Machine Environment

In Screen 9, the programmer uses the standard Emacs command `c-shift-C` (compile definition) to compile the program `REPORT-TIMINGS` so that he can test the program. As part of its normal functioning, this command warns the programmer that the function `MEAN-AND-DEVIATION` has not yet been defined. This is another example of the fact that KBEmacs attempts to build on top of the standard programming environment and use its facilities wherever possible.

Evaluating the Commands Used

Consider the set of knowledge-based commands which were used in order to implement the program `REPORT-TIMINGS`. These commands are summarized below with the direct editing recast as equivalent uses of the knowledge-based command "`Fill`".

```
Define a program REPORT-TIMINGS with a parameter TIMINGS.
Insert a simple-report.
Fill the title with "Report of Reaction Timings (in msec.)".
Fill the enumerator with a list-enumeration of TIMINGS.
Remove the column-headings.
Fill the print-item with a print-out of "~&~5X~8:D".
Fill the summary with (FORMAT REPORT "~2&~{mean:~8:D   (deviation: ~:D)~}"
                                     (MEAN-AND-DEVIATION TIMINGS)).
```

It is interesting to compare these commands with the hypothetical set of directions for an inexperienced assistant programmer (reproduced below). The two sets of directions are quite similar in tone. They share the approach of relying on key terms such as "simple report" and the basic viewpoint of specifying all of the key features of the algorithm.

```
Define a simple report program REPORT-TIMINGS with one parameter, a list of
timings.  Print the title "Report of Reaction Timings (in msec.)".  Print
each timing and then print a summary showing the mean and deviation of the
timings.  Do not print any column headings.
```

However, the knowledge-based commands are significantly more detailed. For example, using KBEmacs the programmer must explicitly refer to the cliche list-enumeration. In addition, he must specify the exact format-strings to use for printing timings and the summary. As will be discussed in Chapter IV, an important future direction for KBEmacs is increasing the capability of the system in order to reduce the detail which has to be specified by the programmer.

```
c-shift-C Compiling REPORT-TIMINGS
The following functions were referenced but do not seem defined:
  MEAN-AND-DEVIATION referenced by REPORT-TIMINGS
```

```
(DEFUN REPORT-TIMINGS (TIMINGS)
  (WITH-OPEN-FILE (REPORT "report.txt" ':OUT)
    (LET* ((DATE (TIME:PRINT-CURRENT-TIME NIL))
           (LINE 66)
           (LIST TIMINGS)
           (PAGE 0)
           (TITLE "Report of Reaction Timings (in msec.)"))
      (FORMAT REPORT "~5%~66: <~A~>~2%~66: <~A~>~%" TITLE DATE)
      (LOOP DO
        (IF (NULL LIST) (RETURN))
        (WHEN (> LINE 62)
          (SETQ PAGE (+ PAGE 1))
          (FORMAT REPORT "~|~%Page:~3D~50: <~A~>~17A~2%" PAGE TITLE DATE)
          (SETQ LINE 3))
        (FORMAT REPORT "~&~5X~8:D" (CAR LIST))
        (SETQ LINE (+ LINE 1))
        (SETQ LIST (CDR LIST)))
      (FORMAT REPORT "~2&~{mean:~8:D   (deviation: ~:D)~}"
              (MEAN-AND-DEVIATION TIMINGS)))))
```

```
KBEmacs (LISP) <KBE.DEMO>TIMINGS.LISP
```

Screen 9

Bottom-Up Implementation

In the next part of the scenario, the programmer proceeds to implement the function `MEAN-AND-DEVIATION`. In doing this, he uses a rather different style of interaction with KBEmacs. Instead of constructing the program in a top-down fashion starting from a top level cliche, he builds up the program from the bottom up by combining various low level cliches. This bottom-up approach is appropriate because there is no interesting top level cliche which captures the overall structure of the desired program.

Directions for a Human Assistant

Consider the kind of directions an expert programmer might give to his assistant. The directions shown below assume that the assistant does not know how to compute the mean and deviation. Although the directions do not refer to any overall high level concept, they make use of lower level concepts such as computing a sum.

```
Define a program MEAN-AND-DEVIATION with one parameter, a list of timings.
Return a list of the mean and deviation of these timings.  The mean is the
sum of the timings divided by the number of timings.  The standard deviation
is the square root of the difference between the second moment and the square
of the mean.  The second moment is the sum of the squares of the timings
divided by the number of timings.
```

It would have been equally plausible to assume that the assistant *did* know how to compute the mean and deviation. In consonance with this, it would also be perfectly plausible to assume that KBEmacs had cliches for computing the mean and deviation. However, the goal of this part of the scenario is to illustrate how low level cliches can be combined together in order to build up a more complex computation. As a result, the scenario assumes that cliches for computing the mean and deviation are not available.

Direct Editing

In order to begin the implementation of the program `MEAN-AND-DEVIATION`, the programmer begins by simply typing in a functional header for the program (see Screen 10). This is often easier than using the knowledge-based command "`Define`".

direct editing

```
(DEFUN MEAN-AND-DEVIATION (TIMINGS)
```

```
KBEmacs (LISP) <KBE.DEMO>TIMINGS.LISP
```

Screen 10

The Cliche Sum

In order to compute the mean, the program MEAN-AND-DEVIATION must compute the sum of the timings. Since this is a stereotyped computation, the programmer asks KBEmacs to insert it into the program (see Screen 11). The programmer specifies the appropriate summation algorithm by combining two cliches: list-enumeration and sum. (Note that this is the same list-enumeration cliche which was used in the implementation of the program REPORT-TIMINGS. The most essential feature of any cliche is that it is a chunk of knowledge which is used again and again in many different situations.)

The cliche sum (shown below) specifies how to accumulate the sum of a series of numbers.

```
(DEFINE-CLICHE SUM
    (PRIMARY-ROLES (NUMBER)
     DESCRIBED-ROLES (NUMBER)
     COMMENT "accumulates the sum of {the number}")
  (LET* ((SUM {0, the zero}))
    (LOOP DO
      (SETQ SUM ({+, the accumulator} SUM {the input number})))
    SUM))
```

The cliche sum is a member of a general class of cliches referred to as accumulators, all of which have essentially the same roles. An input role (here the number) is a sequence of values which is to be combined into some aggregate value. The *accumulator* (here +) is an operator which adds successive input values into the aggregate. The *zero* (here 0) is the zero value corresponding to the accumulator. The return value of the cliche is the sum as a whole. The cliche sum also specifies a name to use for the variable used to accumulate the sum (i.e., SUM).

```
s-X Insert a sum of a list-enumeration of TIMINGS. <end>
```

```
(DEFUN MEAN-AND-DEVIATION (TIMINGS)
  (LET* ((LIST TIMINGS)
         (SUM 0))
    (LOOP DO
      (IF (NULL LIST) (RETURN))
      (SETQ SUM (+ SUM {(CAR LIST), the output element}))
      (SETQ LIST (CDR LIST)))
    SUM))
```

```
KBEmacs (LISP) <KBE.DEMO>TIMINGS.LISP
```

Direct Editing

The programmer continues the implementation of the program `MEAN-AND-DEVIATION` by moving the editing cursor to the end of the program and starting to type an expression specifying what the program should return (see Screen 12) — i.e., a list the first element of which is the mean. The mean in turn is the sum of the timings divided by the number of timings. (It is an idiosyncrasy of the Lisp Machine Lisp dialect used in this scenario that the division function must be written as "`//`".)

Note that the programmer could have proceeded to implement the program `MEAN-AND-DEVIATION` in a more top-down fashion by specifying that the program computed a list of two values the first of which was a division of a sum and a count. However, there would have been no point in doing this because the cliches list (corresponding to the function `LIST`) and division (corresponding to the function `//`) are too trivial to be useful. As mentioned earlier, the goal of KBEmacs is to assist the programmer, not to pedantically force him to do simple things in complex ways.

direct editing

```
(DEFUN MEAN-AND-DEVIATION (TIMINGS)
  (LET* ((LIST TIMINGS)
         (SUM 0))
    (LOOP DO
      (IF (NULL LIST) (RETURN))
      (SETQ SUM (+ SUM {(CAR LIST), the output element}))
      (SETQ LIST (CDR LIST)))
    (LIST (// SUM ))))
```

KBEmacs (LISP) <KBE.DEMO>TIMINGS.LISP

Screen 12

Flexible Interaction

Screen 12 is frozen at the moment where the programmer wants to enter a computation of the number of timings. Since counting the number of timings is a stereotyped operation, Screen 13 shows the programmer using the knowledge-based command "`Insert`" in order to ask KBEmacs to insert the appropriate computation into the program.

The use of the knowledge-based command "`Insert`" in Screen 13 is a good example of the kind of flexible interaction between man and machine that KBEmacs is striving for. In the middle of typing an expression, the programmer issues a request to KBEmacs which is easy to conceive of and even easier to state, but which requires a significant number of changes scattered throughout the program — e.g., in addition to adding computation into the main loop, two new variables are created. Because such commands are available, the programmer can rapidly build up his program by combining together whole algorithms in the order in which they naturally come to mind.

The cliche count determines the length of some enumerated series of elements. Like the cliche sum, count is an accumulation cliche. As will be discussed in detail in Appendix A, the cliche count is particularly interesting in that it depends on a logical input (the elements to be counted), but does not actually use any of these values. As a result, the data flow in Screen 13 does not reveal this dependency.

An interesting feature of the knowledge-based command in Screen 13 is that the input role of the count cliche is specified using the phrase "`the element`" rather than by using another cliche or a literal code fragment. As indicated above, the purpose of output roles is so that they can be referred to in this fashion. It should also be noted that the output role annotation is no longer displayed in Screen 13. Such annotation is suppressed, in the interest of brevity, whenever an output role is directly assigned to a variable since the programmer can use the variable name if he wants to refer to the output value.

Using Common Sense

An important aspect of Screen 13 is that though the changes in the program clearly satisfy the programmer's intent, they do not correspond to the simplest interpretation of the knowledge-based command given. In general, the knowledge-based command "`Insert`" inserts code at the place where the editing cursor is positioned. Given that this is the case, the most straightforward thing for KBEmacs to have done would have been to create code of the following form.

```
(DEFUN MEAN-AND-DEVIATION (TIMINGS)
  (LET* ((COUNT 0)
         (LIST TIMINGS)
         (SUM 0))
    (LOOP DO
      (IF (NULL LIST) (RETURN))
      (SETQ SUM (+ SUM {(CAR LIST), the output element}))
      (SETQ LIST (CDR LIST)))
    (LIST (// SUM (PROGN (LOOP DO (SETQ COUNT (+ COUNT 1))) COUNT)))))
```

There are situations where inserting a sub-loop would be the right thing to do. However, in this situation it seems patently unreasonable. (Among other things, an infinite loop is introduced.) KBEmacs reasons that since the programmer's command refers to a series of values computed in the main program loop, the count cliche should be inserted in that loop with only the final value of the count inserted at the cursor position.

This is an example of a deep problem that KBEmacs only solves in a few limited situations — the use of common sense. As in any discourse, the programmer wants KBEmacs to exhibit common sense. The system tries to do this by rejecting interpretations of knowledge-based commands which are obviously unreasonable. However, KBEmacs is quite limited in the level of its common sense because what may appear to the programmer as only being common sense may in fact require an understanding of programming which is

```
s-X Insert a count of the element. <end>
```

```
(DEFUN MEAN-AND-DEVIATION (TIMINGS)
  (LET* ((COUNT 0)
         (ELEMENT)
         (LIST TIMINGS)
         (SUM 0))
    (LOOP DO
      (IF (NULL LIST) (RETURN))
      (SETQ ELEMENT (CAR LIST))
      (SETQ COUNT (+ COUNT 1))
      (SETQ LIST (CDR LIST))
      (SETQ SUM (+ SUM ELEMENT)))
    (LIST (// SUM COUNT))))
```

```
KBEmacs (LISP) <KBE.DEMO>TIMINGS.LISP
```

Screen 13

much more extensive than the knowledge possessed by KBEmacs.

The Quality of the Code Produced by KBEmacs

As will be discussed in Chapter V, producing reasonably aesthetic program code is the most difficult task KBEmacs has to perform. The code in Screen 13 is a good example of the fact that KBEmacs produces code which is reasonably good, but far from optimal. The code is clearly readable but it has many small problems. For example, KBEmacs is confused by the count cliche into thinking that the output element of the enumerator is used in two places and therefore an unnecessary variable is introduced to hold this value. Further, though the name for the variable created (i.e., `ELEMENT`) is plausible, it is not particularly meaningful in the context of this program. As a final example, within the constraints of data flow, KBEmacs orders the variables bound by the `LET*` alphabetically rather than by using some knowledge-based criterion.

One way to think about the code produced by KBEmacs is to realize that like any code, this code has an identifiable style. From the point of view of the programmer using KBEmacs this is a disadvantage in the sense that he would probably prefer to look at code that is written in his own style rather than in the system's style. However, this disadvantage is counterbalanced by several advantages. First, the programmer is relieved of much of the task of writing the code. If you ask someone else to write some code for you it is only reasonable to expect it to come out largely in their style. Second, since KBEmacs completely recodes a program after each change, the program is always consistent in style rather than taking on a patchwork look after a sequence of modifications. Third, if several programmers are working on a project, than the fact that KBEmacs largely controls the style of the code, might lead to better inter-programmer communication by rendering the code for the project as a whole more stylistically uniform.

Tidying Up the Code Produced by KBEmacs

In Screen 14, the programmer uses the standard Emacs command `m-%` (query replace) to rename the variable `ELEMENT` to `TIMING`. This illustrates the fact that if the programmer does not like the variables chosen by KBEmacs, he can easily change them.

Embedding Cliche References in Program Code

In Screen 14, the programmer finishes the program `MEAN-AND-DEVIATION` by specifying how to compute the deviation. He does this by using standard Emacs commands to directly modify the program code. The expression the programmer inserts illustrates another way in which KBEmacs can be used. Annotation which describes a computation to be performed in terms of cliches can be inserted directly in a program. KBEmacs will eventually replace this annotation with the appropriate code. This is equivalent to using the knowledge-based command "`Insert`", but can be more readable.

The first piece of annotation specifies how to compute a sum of the squares of the individual timings. It uses the cliches sum and squaring. The second piece of annotation is used to refer to the value of the mean. The phrase "`a use of`" makes it possible to refer to a quantity even when it is not stored in a variable. Without this capability, the programmer would have to introduce a variable himself. The phrase "`the first //`" identifies the value as the output of the first (in execution order) division operation.

m-c-H c-U m-% ***replace some occurrences of***: ELEMENT ***with***: TIMING
direct editing

```
(DEFUN MEAN-AND-DEVIATION (TIMINGS)
  (LET* ((COUNT 0)
         (TIMING)
         (LIST TIMINGS)
         (SUM 0))
    (LOOP DO
      (IF (NULL LIST) (RETURN))
      (SETQ TIMING (CAR LIST))
      (SETQ COUNT (+ COUNT 1))
      (SETQ LIST (CDR LIST))
      (SETQ SUM (+ SUM TIMING)))
    (LIST (// SUM COUNT)
          (ISQRT (- (// {a sum of a squaring of TIMING} COUNT)
                    (^ {a use of the first //} 2)))))
```

Replacing Embedded Cliche References With Code

In Screen 15, the programmer tells KBEmacs that the program has been completed by using the command `s-<end>`. The major effect this has is to cause KBEmacs to analyze the direct editing done by the programmer in the last screen and to replace the annotation introduced with ordinary program code. As part of this, a second summation is inserted into the main loop. This summation computes the sum of the squares of the timings. Due to the fact that the variable name `SUM` is already used in the program, KBEmacs uses the name `SUM1` for the accumulation variable of this second summation. In addition, a variable is introduced so that the value of the mean can be used twice. The name `X` is used for this variable because KBEmacs has no basis for determining a better name to use. Finally, in order to maintain its stylistic constraints, KBEmacs reorders the bound variables so that they are alphabetical. (They got out of order due the the variable renaming in Screen 14.)

s-<end> ***Finish editing the function MEAN-AND-DEVIATION. <end>***

```
(DEFUN MEAN-AND-DEVIATION (TIMINGS)
  (LET* ((COUNT 0)
         (LIST TIMINGS)
         (SUM 0)
         (SUM1 0)
         (TIMING)
         (X))
    (LOOP DO
      (IF (NULL LIST) (RETURN))
      (SETQ TIMING (CAR LIST))
      (SETQ SUM1 (+ SUM1 (^ TIMING 2)))
      (SETQ COUNT (+ COUNT 1))
      (SETQ LIST (CDR LIST))
      (SETQ SUM (+ SUM TIMING)))
    (SETQ X (// SUM COUNT))
    (LIST X (ISQRT (- (// SUM1 COUNT) (^ X 2))))))
```

KBEmacs (LISP) <KBE.DEMO>TIMINGS.LISP

Tidying Up the Code Produced by KBEmacs

Looking at the result produced in Screen 15, the programmer decides to change the variable names chosen by KBEmacs so that they are more mnemonic. He does this using standard editing commands in Screen 16 and then compiles the program so that it can be tested.

Evaluating the Commands Used

The KBEmacs commands used in this part of the scenario can by nicely summarized by supposing that all of the cliches were referred to with embedded annotation. (If embedded annotation had actually been used as shown below, it would have resulted in a correctly running program. However, the program would have been quite inefficient because there would have been four separate list-enumerations and two separate counts. KBEmacs is not capable of automatically determine that these computations should be shared. Achieving this sharing is an important reason why the scenario above took the form it did.)

```
(DEFUN MEAN-AND-DEVIATION (TIMINGS)
  (LIST (// {a sum of a list-enumeration of TIMINGS}
            {a count of a list-enumeration of TIMINGS})
        (ISQRT (- (// {a sum of a squaring of a list-enumeration of TIMINGS}
                      {a count of a list-enumeration of TIMINGS})
                  (^ {a use of the first //} 2)))))
```

It can be seen that the knowledge-based commands used have much the same form as the set of directions for a human assistant hypothesized at the beginning of this section.

```
Define a program MEAN-AND-DEVIATION with one parameter, a list of timings.
Return a list of the mean and deviation of these timings.  The mean is the
sum of the timings divided by the number of timings.  The standard deviation
is the square root of the difference between the second moment and the square
of the mean.  The second moment is the sum of the squares of the timings
divided by the number of timings.
```

```
m-c-H c-U m-% replace some occurrences of: SUM1 with: SUM-SQUARES
m-c-H c-U m-% replace some occurrences of: X with: MEAN
s-<end> Finish editing the function MEAN-AND-DEVIATION. <end>
c-shift-C Compiling MEAN-AND-DEVIATION
```

```
(DEFUN MEAN-AND-DEVIATION (TIMINGS)
  (LET* ((COUNT 0)
         (LIST TIMINGS)
         (MEAN)
         (SUM 0)
         (SUM-SQUARES 0)
         (TIMING))
    (LOOP DO
      (IF (NULL LIST) (RETURN))
      (SETQ TIMING (CAR LIST))
      (SETQ SUM-SQUARES (+ SUM-SQUARES (^ TIMING 2)))
      (SETQ COUNT (+ COUNT 1))
      (SETQ LIST (CDR LIST))
      (SETQ SUM (+ SUM TIMING)))
    (SETQ MEAN (// SUM COUNT))
    (LIST MEAN (ISQRT (- (// SUM-SQUARES COUNT) (^ MEAN 2))))))
```

KBEmacs (LISP) <KBE.DEMO>TIMINGS.LISP

Testing

In Screen 17, the programmer uses the standard Lisp Machine context switching command <select> L in order to temporarily switch to the Lisp execution environment. He then defines a test program TEST-REPORT-TIMINGS and tests the program REPORT-TIMINGS. The program DISPLAY-REPORT (see Appendix B) displays the report produced. In order to save screen space the program DISPLAY-REPORT abbreviates groups of blank lines with a single line of the form "<*n* blank lines>". Page breaks are represented by "<page>". Screen 17 is yet another example of the fact that KBEmacs is integrated with the Lisp Machine environment in such a way that the programmer can use all of the normal Lisp Machine facilities.

```
<select> L
```

```
(DEFUN TEST-REPORT-TIMINGS ()
  (REPORT-TIMINGS '(10041 9315 10722 11473 10834 11076 10447 10658 9529
                    9041 10452 11137 10351 10384 10474 11706 9592 10685))
  (DISPLAY-REPORT))
TEST-REPORT-TIMINGS
(TEST-REPORT-TIMINGS)
<5 blank lines>
              Report of Reaction Timings (in msec.)

                         5/27/85 12:16:39
<page>
Page:  1      Report of Reaction Timings (in msec.)      5/27/85 12:16:39

       10,041
        9,315
       10,722
       11,473
       10,834
       11,076
       10,447
       10,658
        9,529
        9,041
       10,452
       11,137
       10,351
       10,384
       10,474
       11,706
        9,592
       10,685

mean:  10,439   (deviation:713)
```

Lisp Listener

Screen 17

Modification and Documentation

Although KBEmacs focuses primarily on the task of program construction, the basic capabilities of the system provide considerable leverage on other aspects of the programming process. In order to illustrate this fact, KBEmacs supports a few commands which are directed explicitly toward the tasks of program modification and documentation.

The Knowledge-Based Command "Replace"

Looking at the test output produced in Screen 17, the programmer decides that the report would look better if the timings were printed out in a tabular form rather than one to a line. In order to perform this modification, the programmer first switches back into the editor and then uses the standard Emacs command `m-.` (goto definition) in order to move the editing cursor to the definition of the function `REPORT-TIMINGS` (see Screen 18).

Having located the function `REPORT-TIMINGS` the programmer uses the knowledge-based command "`Replace`" in order to change the print-item from a print-out to a tabularized-print-out. The command "`Replace`" removes the old code filling the specified role and replaces it with the new computation specified. The essential underpinning of the command is the fact that KBEmacs remembers where the various roles in a program are even after they have been filled in.

The Cliche Tabularized-Print-Out

The cliche tabularized-print-out (shown below) is similar to the cliche print-out in that it has two primary roles, a format-string and an object to print. However, unlike the cliche print-out, the cliche tabularized-print-out assumes that the format-string will not force a newline, and prints as many objects as possible on each line. Before printing out an object the cliche tests to see whether there is enough room at the end of the current line in order to print it. If there is not enough room, then a newline is inserted (and the line number is incremented). The Lisp Machine Lisp function `CHARPOS` returns the output character position on the current line. The role maximum-charpos is derived by a constraint based on the assumption that only 75 characters can be printed on a line. The function `SIZE-IN-CHARACTERS` is capable of analyzing simple format control strings and determining the maximum number of characters that will be printed.

```
(DEFINE-CLICHE TABULARIZED-PRINT-OUT
    (PRIMARY-ROLES (FORMAT-STRING ITEM)
     DESCRIBED-ROLES (FORMAT-STRING ITEM NUMBER-OF-COLUMNS)
     COMMENT "prints out {the item} in columns"
     CONSTRAINTS
      ((DEFAULT {the format-string} "~15A")
       (DERIVED {the maximum-charpos}
                (- 75 (SIZE-IN-CHARACTERS {the format-string})))))
  (WHEN (> (CHARPOS REPORT) {the maximum-charpos})
    (FORMAT REPORT "~&")
    (SETQ LINE (+ LINE 1)))
  (FORMAT REPORT {the format-string} {the input item}))
```

In Screen 18, the format-string specified prints out 10 characters. As a result, the maximum-charpos is set at `65`. The line-limit role of the simple-report remains fixed at `62` because the tabularized-print-out can print at most one output line.

```
<select> E
m-. Edit definition of REPORT-TIMINGS
s-X Replace the print-item with a tabularized-print-out of "~10:D". <end>
s-<end> Finish editing the function REPORT-TIMINGS. <end>
c-shift-C Compiling REPORT-TIMINGS
```

```
(DEFUN REPORT-TIMINGS (TIMINGS)
  (WITH-OPEN-FILE (REPORT "report.txt" ':OUT)
    (LET* ((DATE (TIME:PRINT-CURRENT-TIME NIL))
           (LINE 66)
           (LIST TIMINGS)
           (PAGE 0)
           (TITLE "Report of Reaction Timings (in msec.)"))
      (FORMAT REPORT "~5%~66:<~A~>~2%~66:<~A~>~%" TITLE DATE)
      (LOOP DO
        (IF (NULL LIST) (RETURN))
        (WHEN (> LINE 62)
          (SETQ PAGE (+ PAGE 1))
          (FORMAT REPORT "~|~%Page:~3D~50:<~A~>~17A~2%" PAGE TITLE DATE)
          (SETQ LINE 3))
        (WHEN (> (CHARPOS REPORT) 65)
          (FORMAT REPORT "~&")
          (SETQ LINE (+ LINE 1)))
        (FORMAT REPORT "~10:D" (CAR LIST))
        (SETQ LIST (CDR LIST)))
      (FORMAT REPORT "~2&~{mean:~8:D   (deviation: ~:D)~}"
              (MEAN-AND-DEVIATION TIMINGS)))))
```

Retesting

After recompiling the program REPORT-TIMEBOX (see Screen 18), the programmer switches to the Lisp execution environment and retests the program (see Screen 19).

<select> L

```
(TEST-REPORT-TIMINGS)
<5 blank lines>
                Report of Reaction Timings (in msec.)

                            5/27/85 12:17:07
<page>
Page:  1        Report of Reaction Timings (in msec.)      5/27/85 12:17:07

    10,041     9,315    10,722    11,473    10,834    11,076    10,447
    10,658     9,529     9,041    10,452    11,137    10,351    10,384
    10,474    11,706     9,592    10,685

mean:  10,439    (deviation: 713)
```

Lisp Listener

The Knowledge-Based Command "Comment"

In Screen 20, the programmer switches back to the editor and asks KBEmacs to generate a comment for the function `REPORT-TIMINGS`. This is an example of one way in which KBEmacs can use the information it maintains about the structure of a program in order to assist a programmer in understanding the program. The knowledge-based command "Comment" (here triggered by the abbreviated command `s-;`) creates a summary comment describing a program and inserts this comment into the editor buffer.

The comment is in the form of an outline. The first line specifies the top level cliche in the program. The subsequent entries describe how the major roles in this cliche have been filled. The comment is constructed based on the cliches that were used to create the program. The `DESCRIBED-ROLES` declaration from the definition of the top level cliche specifies which roles to describe and what order to describe them in.

Each role is described in one of four ways. If the role has been removed, then it is reported as missing (e.g., the column-headings). If the role is filled by a cliche then this cliche is named (e.g., list-enumeration). Further, a brief one line description showing how the roles of this sub-cliche are filled in is included. (This description is generated from the `COMMENT` declaration from the definition of the sub-cliche.) If a role is filled with non-cliched computation which is short enough to fit on a single line then the corresponding code is displayed (e.g., the title); otherwise, the computation is simply reported to be idiosyncratic (e.g., the summary). The individual lines of the comment are written in the same style as the knowledge-based commands. In particular, variable names and code fragments are rendered in upper case.

Evaluating the Comment Produced

The comment generation capability currently supported by KBEmacs is only intended as an illustration of the kind of comment that could be produced. There are many other kinds of comments containing either more or less information that could just as well have been produced. For example, KBEmacs could easily include a description of the inputs and outputs of the program in the comment. The form of comment shown was chosen because it contains a significant amount of high level information which is not explicit in the program code. As a result, it should be of genuine assistance to a person who is trying to understand the program.

A key feature of the comment is that, since it is generated from the knowledge underlying the program, it is guaranteed to be complete and correct. In contrast, much of the program documentation one typically encounters has been rendered obsolete by subsequent program modifications. Although it is not currently supported, it would be relatively easy for KBEmacs to generate a new program comment every time a program was modified. Using this approach it might be possible for such commentary to augment (or even partially replace) program code as the means of communication between KBEmacs and the programmer.

The Knowledge-Based Command "Highlight"

Another way in which KBEmacs can assist a programmer in understanding a program is to point out various parts of the program. In Screen 20, the programmer asks to see the summary role. The knowledge-based command "Highlight" uses standard editor highlighting (i.e., underlining) in order to show the programmer a particular part of a program. In order to support this command, KBEmacs maintains a mapping which shows what pieces of the text for a program correspond to each role in the program.

```
<select> E
s-; Comment the function REPORT-TIMINGS. <end>
s-X Highlight the summary. <end>
```

```
;;; The function REPORT-TIMINGS is a simple-report.
;;;   The file-name is "report.txt".
;;;   The title is "Report of Reaction Timings (in msec.)".
;;;   The enumerator is a list-enumeration.
;;;      It enumerates the elements of TIMINGS.
;;;   There are no column-headings.
;;;   The print-item is a tabularized-print-out.
;;;      It prints out (CAR LIST) in columns.
;;;   The summary is an idiosyncratic computation.

(DEFUN REPORT-TIMINGS (TIMINGS)
  (WITH-OPEN-FILE (REPORT "report.txt" ':OUT)
    (LET* ((DATE (TIME:PRINT-CURRENT-TIME NIL))
           (LINE 66)
           (LIST TIMINGS)
           (PAGE 0)
           (TITLE "Report of Reaction Timings (in msec.)"))
      (FORMAT REPORT "~5%~66: <~A~>~2%~66: <~A~>~%" TITLE DATE)
      (LOOP DO
        (IF (NULL LIST) (RETURN))
        (WHEN (> LINE 62)
          (SETQ PAGE (+ PAGE 1))
          (FORMAT RFPORT "~|~%Page:~3D~50: <~A~>~17A~2%" PAGE TITLE DATE)
          (SETQ LINE 3))
        (WHEN (> (CHARPOS REPORT) 65)
          (FORMAT REPORT "~&")
          (SETQ LINE (+ LINE 1)))
        (FORMAT REPORT "~10:D" (CAR LIST))
        (SETQ LIST (CDR LIST)))
      (FORMAT REPORT "~2&~{mean:~8:D   (deviation: ~:D)~}"
              (MEAN-AND-DEVIATION TIMINGS)))))
```

Defining a Cliche

In the remainder of the Lisp scenario, the programmer defines a new cliche. This part of the scenario seeks to demonstrate two main points. First, defining a new cliche is no more difficult than defining a program. It is expected that programmers will define their own idiosyncratic cliches as readily as they define new subroutines. As with subroutines, the key difficulty resides in deciding exactly what cliches should be defined.

Second, this part of the scenario demonstrates a third style of program construction — the lateral modification of almost-right cliches. In this style the programmer begins with a cliche which embodies much of what he wants and then fixes it up in order to get exactly what he wants. Any real programming situation involves a combination of the top-down, bottom-up, and lateral modification styles of implementation.

Directions for a Human Assistant

In the scenario below, the programmer defines a new cliche report-with-subheadings. This cliche is similar to the cliche simple-report except that it assumes that the items to be reported are divided into groups and makes it possible for the user to specify the printing of a subheading before each group. The directions shown below illustrate how an expert programmer might tell an assistant programmer what to do. These directions are couched in terms of lateral modification. They rely on an understanding of the terms "simple report" and "group detector" (see the description of Screen 22 for a discussion of the latter term).

```
Define a cliche report-with-subheadings.  This cliche should have all of the
features of a simple-report.  In addition, the cliche should have two new
roles: the group-test and the subheading.  The group-test should be the test
of a group detector.  The subheading role should be used to print a
subheading each time the start of a new group is detected.
```

The Knowledge-Based Command "`Copy`"

In order to define the cliche report-with-subheadings, the programmer reads in the file of cliches and then uses the knowledge-based command "`Define`" in order to create an initial empty definition. He then uses the knowledge-based command "`Copy`" in order to create a copy of the definition of the cliche simple-report (see Screen 21). The difference between the knowledge-based commands "`Copy`" and "`Insert`" is that while "`Insert`" creates an instance of a cliche, "`Copy`" copies over all of the commentary about a cliche, creating an analogous cliche definition. The knowledge-based command "`Copy`" is provided specifically in order to facilitate the definition of cliches which are similar to predefined cliches.

c-X c-F *Find file* <KBE.DEMO>CLICHE-LIBRARY.LISP
s-X Define a cliche REPORT-WITH-SUBHEADINGS. <end>
s-X Copy the cliche SIMPLE-REPORT to the cliche REPORT-WITH-SUBHEADINGS. <end>
cursor motion

```
(DEFINE-CLICHE REPORT-WITH-SUBHEADINGS
    (PRIMARY-ROLES (ENUMERATOR PRINT-ITEM SUMMARY)
     DESCRIBED-ROLES (FILE-NAME TITLE ENUMERATOR COLUMN-HEADINGS
                      PRINT-ITEM SUMMARY)
     COMMENT "prints a report of {the input structure of the enumerator}"
     CONSTRAINTS
       ((DEFAULT {the file-name} "report.txt")
        (DERIVED {the line-limit}
                 (- 65
                    (SIZE-IN-LINES {the print-item})
                    (SIZE-IN-LINES {the summary})))))
  (WITH-OPEN-FILE (REPORT {the file-name} ':OUT)
    (LET* ((DATE (TIME:PRINT-CURRENT-TIME NIL))
           (LINE 66)
           (PAGE 0)
           (TITLE {the title})
           (DATA {the input structure of the enumerator}))
      (FORMAT REPORT "~5%~66: <~A~>~2%~66: <~A~>~%" TITLE DATE)
      (LOOP DO
        (IF ({the empty-test of the enumerator} DATA) (RETURN))
        (WHEN (> LINE {the line-limit})
          (SETQ PAGE (+ PAGE 1))
          (FORMAT REPORT "~|~%Page:~3D~50: <~A~>~17A~2%" PAGE TITLE DATE)
          (SETQ LINE 3)
          ({the column-headings} {REPORT, modified} {LINE, modified}))
        ({the print-item} {REPORT, modified}
                          {LINE, modified}
                          ({the element-accessor of the enumerator} DATA))
        (SETQ DATA ({the step of the enumerator} DATA)))
      ({the summary} {REPORT, modified}))))
```

KBEmacs (LISP) <KBE.DEMO>CLICHE-LIBRARY.LISP

The Cliche Group-detector

Before continuing with the definition of the cliche report-with-subheadings it is important to take a look at the cliche group-detector (shown below). The purpose of this cliche is to identify groups in a series of items. For example consider the series of digits 2 2 8 6 6 6. This series is composed of groups of repeated digits (i.e., two 2s, one 8, and three 6s). The cliche group-detector could be used to signal the start of each group in the series.

The cliche group-detector has three roles: the input item, the test, and the output flag. The input item is the series to be tested. The test is a function which determines when a new group is beginning. The cliche assumes that the start of a new group can be detected by comparing the current item with its predecessor. The test is intended to be a function of two arguments (the prior item and the current item) which returns T if and only if the current item is the start of a new group. The output flag is a series of truth values specifying whether or not the corresponding items start new groups. For example, if the input items were the series of digits shown above and the test were the function =, then the output flag would be the series T NIL T T NIL NIL.

```
(DEFINE-CLICHE GROUP-DETECTOR
    (PRIMARY-ROLES (ITEM TEST)
     DESCRIBED-ROLES (ITEM TEST)
     COMMENT "locates the beginning of each group in {the item}")
  (LET* ((DATUM)
         (UNIQUE-VALUE (NCONS NIL))
         (PREVIOUS UNIQUE-VALUE))
    (LOOP DO
      (SETQ DATUM {the input item})
      {(OR (EQ PREVIOUS UNIQUE-VALUE) ({the test} PREVIOUS DATUM)),
       the output flag}
      (SETQ PREVIOUS DATUM))))
```

The main complexity in the cliche group-detector comes from handling the boundary conditions at the first input item. The problem is that this item has no predecessor. It is however, guaranteed to begin a new group — the first group. To deal with this, the cliche initializes the variable PREVIOUS with a unique value — one which cannot be EQ to any other data item. By comparing the value of the variable with this unique value, the cliche is able to insure that the test is only applied to items after the first one. In Lisp, this is an efficient and compact algorithm.

One way to look at the cliche group-detector is as an abstraction shift which connects the simple idea of a function which detects groups by comparing successive items in a series into an algorithm which actually works. The key virtue of the cliche is that it relieves the programmer from having to worry about the detailed code — which is, in truth, rather obscure.

Note that it is tempting to simplify the code by initializing the variable PREVIOUS to NIL. However, this would introduce a hidden bug which would appear only when one of the data items was NIL. This is an example of the fact that one has to be careful when defining a cliche. However, this carefulness pays off in enhanced program reliability when the cliche is used.

Lateral Modification

The difference between the cliches simple-report and report-with-subheadings is that before printing out each item, the latter cliche must potentially print out a subheading. In order to introduce this added computation, the programmer first positions the editing cursor before the print-item role (see Screen 21), and then inserts a group-detector (see Screen 22). The code produced is another example of how a simple command to KBEmacs can cause a number of non-local modifications to a program. Note in particular that the insertion of the cliche group-detector forces the introduction of a variable DATUM to hold the value of the

```
s-X Insert a group-detector of the element-accessor. <end>
```

```
(DEFINE-CLICHE REPORT-WITH-SUBHEADINGS
    (PRIMARY-ROLES (ENUMERATOR PRINT-ITEM SUMMARY)
     DESCRIBED-ROLES (FILE-NAME TITLE ENUMERATOR COLUMN-HEADINGS
                      PRINT-ITEM SUMMARY)
     COMMENT "prints a report of {the input structure of the enumerator}"
     CONSTRAINTS
       ((DEFAULT {the file-name} "report.txt")
        (DERIVED {the line-limit}
                 (- 65
                    (SIZE-IN-LINES {the print-item})
                    (SIZE-IN-LINES {the summary})))))
  (WITH-OPEN-FILE (REPORT {the file-name} ':OUT)
    (LET* ((DATE (TIME:PRINT-CURRENT-TIME NIL))
           (DATUM)
           (LINE 66)
           (PAGE 0)
           (UNIQUE-VALUE (NCONS NIL))
           (PREVIOUS UNIQUE-VALUE)
           (TITLE {the title})
           (DATA {the input structure of the enumerator}))
      (FORMAT REPORT "~5%~66:<~A~>~2%~66:<~A~>~%" TITLE DATE)
      (LOOP DO
        (IF ({the empty-test of the enumerator} DATA) (RETURN))
        (SETQ DATUM ({the element-accessor of the enumerator} DATA))
        (WHEN (> LINE {the line-limit})
          (SETQ PAGE (+ PAGE 1))
          (FORMAT REPORT "~|~%Page:~3D~50:<~A~>~17A~2%" PAGE TITLE DATE)
          (SETQ LINE 3)
          ({the column-headings} {REPORT, modified} {LINE, modified}))
        ({the print-item} {REPORT, modified} {LINE, modified} DATUM)
        {(OR (EQ PREVIOUS UNIQUE-VALUE) ({the test} PREVIOUS DATUM)),
          the output flag}
        (SETQ DATA ({the step of the enumerator} DATA))
        (SETQ PREVIOUS DATUM))
      ({the summary} {REPORT, modified}))))
```

current element being enumerated.

The cliche simple-report is being used in a completely different way in this part of the scenario than the way it was used in the implementation of the program REPORT-TIMINGS. When implementing REPORT-TIMINGS, the cliche was used because the desired program was an instance of this cliche. Here, the cliche is used merely because it contains a lot of structure which is desired and therefore it is convenient to use the cliche as a starting point. It should be noted, however, that once the group-detector has been introduced, the code is no longer an instance of a simple-report and the programmer must accept full responsibility for deciding what parts of the cliche are still appropriate.

An interesting aspect of Screen 22 is that the group-detector does not end up before the print-item. This is another example of the limitations of the ability of KBEmacs to generate aesthetic program code. KBEmacs orders the statements in a program solely based on data flow and control flow constraints. In Screen 22, there is no data flow or control flow reason why the group-detector must be placed either before or after the print-item. In such situations, the statement order chosen by KBEmacs is essentially arbitrary. Unfortunately, KBEmacs is not able to recognize the cursor position in Screen 21 as, in effect, a comment signaling the fact that the programmer intends the group-detector to appear in that position. Rather, KBEmacs only uses the cursor position to determine the data flow and control flow environment of the insertion.

Inserting the Subheading Role

In Screen 23, the programmer uses direct editing in order to specify the printing of subheadings. He creates a role corresponding to the printing of subheadings by simply typing in {...} annotation describing the role. Like the print-item, this role is expected to side-effect the REPORT file, update the line number, and use the current element being enumerated.

In order to specify when the subheadings should be printed, the programmer uses a conditional predicated on the output of the group-detector. At first glance it may seem odd that the programmer can refer to the group-detector even though the conditional precedes the group-detector. This is allowed in order to give the programmer added flexibility. The programmer can refer to a following quantity unless the data flow specifies that the quantity is required to follow.

direct editing

```
(DEFINE-CLICHE REPORT-WITH-SUBHEADINGS
    (PRIMARY-ROLES (ENUMERATOR PRINT-ITEM SUMMARY)
     DESCRIBED-ROLES (FILE-NAME TITLE ENUMERATOR COLUMN-HEADINGS
                      PRINT-ITEM SUMMARY)
     COMMENT "prints a report of {the input structure of the enumerator}"
     CONSTRAINTS
       ((DEFAULT {the file-name} "report.txt")
        (DERIVED {the line-limit}
                 (- 65
                    (SIZE-IN-LINES {the print-item})
                    (SIZE-IN-LINES {the summary})))))
  (WITH-OPEN-FILE (REPORT {the file-name} ':OUT)
    (LET* ((DATE (TIME:PRINT-CURRENT-TIME NIL))
           (DATUM)
           (LINE 66)
           (PAGE 0)
           (UNIQUE-VALUE (NCONS NIL))
           (PREVIOUS UNIQUE-VALUE)
           (TITLE {the title})
           (DATA {the input structure of the enumerator}))
      (FORMAT REPORT "~5%~66: <~A~>~2%~66: <~A~>~%" TITLE DATE)
      (LOOP DO
        (IF ({the empty-test of the enumerator} DATA) (RETURN))
        (SETQ DATUM ({the element-accessor of the enumerator} DATA))
        (WHEN (> LINE {the line-limit})
          (SETQ PAGE (+ PAGE 1))
          (FORMAT REPORT "~|~%Page:~3D~50: <~A~>~17A~2%" PAGE TITLE DATE)
          (SETQ LINE 3)
          ({the column-headings} {REPORT, modified} {LINE, modified}))
        (IF {a use of the flag}
            ({the subheading} {REPORT, modified} {LINE, modified} DATUM))
        ({the print-item} {REPORT, modified} {LINE, modified} DATUM)
        {(OR (EQ PREVIOUS UNIQUE-VALUE) ({the test} PREVIOUS DATUM)),
          the output flag}
        (SETQ DATA ({the step of the enumerator} DATA))
        (SETQ PREVIOUS DATUM))
      ({the summary} {REPORT, modified}))))
```

KBEmacs (LISP) <KBE.DEMO>CLICHE-LIBRARY.LISP

Screen 23

The Knowledge-Based Command "`Analyze`"

The knowledge-based command "`Analyze`" tells KBEmacs to analyze the effects of direct editing done by the programmer without initiating any other action. In Screen 24, this causes the annotation referring to the group-detector to be replaced by the corresponding code. Note that the group-detector is now positioned before the print-item because it is required to be there by the data flow. (The control flow in the conditional forces the group-detector to be before the subheading, and the fact that the subheading side-effects the report file forces the subheading to be before the print-item.)

s-A ***Analyze the cliche REPORT-WITH-SUBHEADINGS. <end>***

```
(DEFINE-CLICHE REPORT-WITH-SUBHEADINGS
    (PRIMARY-ROLES (ENUMERATOR PRINT-ITEM SUMMARY)
     DESCRIBED-ROLES (FILE-NAME TITLE ENUMERATOR COLUMN-HEADINGS
                      PRINT-ITEM SUMMARY)
     COMMENT "prints a report of {the input structure of the enumerator}"
     CONSTRAINTS
       ((DEFAULT {the file-name} "report.txt")
        (DERIVED {the line-limit}
                 (- 65
                    (SIZE-IN-LINES {the print-item})
                    (SIZE-IN-LINES {the summary})))))
  (WITH-OPEN-FILE (REPORT {the file-name} ':OUT)
    (LET* ((DATE (TIME:PRINT-CURRENT-TIME NIL))
           (DATUM)
           (LINE 66)
           (PAGE 0)
           (UNIQUE-VALUE (NCONS NIL))
           (PREVIOUS UNIQUE-VALUE)
           (TITLE {the title})
           (DATA {the input structure of the enumerator}))
      (FORMAT REPORT "~5%~66: <~A~>~2%~66: <~A~>~%" TITLE DATE)
      (LOOP DO
        (IF ({the empty-test of the enumerator} DATA) (RETURN))
        (SETQ DATUM ({the element-accessor of the enumerator} DATA))
        (WHEN (> LINE {the line-limit})
          (SETQ PAGE (+ PAGE 1))
          (FORMAT REPORT "~|~%Page:~3D~50: <~A~>~17A~2%" PAGE TITLE DATE)
          (SETQ LINE 3)
          ({the column-headings} {REPORT, modified} {LINE, modified}))
        (IF {XOR (EQ PREVIOUS UNIQUE-VALUE) ({the test} PREVIOUS DATUM)),
             the output flag}
            ({the subheading} {REPORT, modified} {LINE, modified} DATUM))
        ({the print-item} {REPORT, modified} {LINE, modified} DATUM)
        (SETQ DATA ({the step of the enumerator} DATA))
        (SETQ PREVIOUS DATUM))
      ({the summary} {REPORT, modified}))))
```

Editing the Cliche Declarations

In Screen 25, the programmer finishes the cliche report-with-subheadings by appropriately updating the declarations in the cliche definition. First, he renames the test of the group-detector to be the role group-test. He then specifies that the subheading and group-test should be described roles. Next, he updates the constraint deriving the line-limit so that it accounts for the subheading as well as the print-item and the summary. Finally, he deletes the commentary describing the flag output of the group-detector because he does not want this to be a role of the cliche report-with-subheadings.

After finishing the cliche, the programmer uses the knowledge-based command "`Finish`" in order to signal this fact to KBEmacs. When operating on a cliche definition, the knowledge-based command "`Finish`" does not complain about unfilled roles, or remove output role annotation because these things are an essential part of cliches.

Finally, the programmer uses the standard Emacs command `c-shift-C` (compile definition) in order to compile the cliche. The act of compiling a cliche makes it available to KBEmacs. From this point on, the programmer can use the cliche report-with-subheadings just like any other cliche.

Evaluating the Commands Used

A comparison of the commands which were used in order to implement the cliche report-with-subheadings i.e.:

```
Define a cliche REPORT-WITH-SUBHEADINGS.
Copy to the cliche REPORT-WITH-SUBHEADINGS the cliche SIMPLE-REPORT.
Insert a group-detector of the element-accessor.
```
direct editing

with the directions that might have been given to a human assistant i.e.:

```
Define a cliche report-with-subheadings.  This cliche should have all of the
features of a simple-report.  In addition, the cliche should have two new
roles: the group-test and the subheading.  The group-test should be the test
of a group detector.  The subheading role should be used to print a
subheading each time the start of a new group is detected.
```

shows that they have much of the same character. However, in this case, the knowledge-based commands only support the top level parts of the directions. The lower level changes are done by means of direct editing.

direct editing

s-<end> ***Finish editing the cliche REPORT-WITH-SUBHEADINGS. <end>***
c-shift-C ***Compiling REPORT-WITH-SUBHEADINGS***

```
(DEFINE-CLICHE REPORT-WITH-SUBHEADINGS
    (PRIMARY-ROLES (ENUMERATOR PRINT-ITEM SUMMARY)
     DESCRIBED-ROLES (FILE-NAME TITLE ENUMERATOR COLUMN-HEADINGS
                      PRINT-ITEM SUBHEADING GROUP-TEST SUMMARY)
     COMMENT "prints a report of {the input structure of the enumerator}"
     CONSTRAINTS
       ((DEFAULT {the file-name} "report.txt")
        (DERIVED {the line-limit}
                 (- 65
                    (SIZE-IN-LINES {the print-item})
                    (SIZE-IN-LINES {the summary})
                    (SIZE-IN-LINES {the subheading})))))
  (WITH-OPEN-FILE (REPORT {the file-name} ':OUT)
    (LET* ((DATE (TIME:PRINT-CURRENT-TIME NIL))
           (DATUM)
           (LINE 66)
           (PAGE 0)
           (UNIQUE-VALUE (NCONS NIL))
           (PREVIOUS UNIQUE-VALUE)
           (TITLE {the title})
           (DATA {the input structure of the enumerator}))
      (FORMAT REPORT "~5%~66:<~A~>~2%~66:<~A~>~%" TITLE DATE)
      (LOOP DO
        (IF ({the empty-test of the enumerator} DATA) (RETURN))
        (SETQ DATUM ({the element-accessor of the enumerator} DATA))
        (WHEN (> LINE {the line-limit})
          (SETQ PAGE (+ PAGE 1))
          (FORMAT REPORT "~|~%Page:~3D~50:<~A~>~17A~2%" PAGE TITLE DATE)
          (SETQ LINE 3)
          ({the column-headings} {REPORT, modified} {LINE, modified}))
        (IF (OR (EQ PREVIOUS UNIQUE-VALUE)
                ({the group-test} PREVIOUS DATUM))
            ({the subheading} {REPORT, modified} {LINE, modified} DATUM))
        ({the print-item} {REPORT, modified} {LINE, modified} DATUM)
        (SETQ DATA ({the step of the enumerator} DATA))
        (SETQ PREVIOUS DATUM))
      ({the summary} {REPORT, modified}))))
```

KBEmacs (LISP) <KBE.DEMO>CLICHE-LIBRARY.LISP Screen 25

III - Ada Scenario

An important aspect of KBEmacs is that both the ideas behind the system and the actual implementation of the system are substantially programming language independent. As will be discussed in Chapter V, to extend KBEmacs so that it can be used to operate on a given programming language, one need only do two things: give the system an understanding of the syntax and semantics of the language in question, and provide a library of cliches which are appropriate for writing programs in that language. This chapter demonstrates the language independence of KBEmacs by showing the system being used to construct several Ada programs. Although this chapter does not assume that the reader knows the details of Ada, it does assume a familiarity with Algol-like languages in general. The reference manual [Ada 83] describes Ada in detail.

Another purpose of this chapter is to show that KBEmacs can be used to operate on programs of realistic size and complexity. Earlier scenarios (whether proposed [Rich Shrobe 78] or real [Waters 82a]) have focused on simple programs only 10 to 20 lines long. Although this was done primarily in the interest of brevity, it naturally brought up the question of whether the system could be applied to large programs. The last section of this chapter shows the construction of a 110 line Ada program. It could be argued that programs should not be larger than this.

The scenario in this chapter is also used to demonstrate a number of points which are brought out better by the language Ada than by Lisp. For example, unlike Lisp, data declarations are an important part of any Ada program. This can be a source of programmer frustration because, though much of the information in data declarations is redundant with the rest of the program, it must all be independently specified. This is an example of a task for which KBEmacs is ideally suited. KBEmacs can automatically generate most of the data declarations in a typical program.

Ada Cliches

In the scenario below, as in the Lisp scenario in the last chapter, cliches are central to the communication between the programmer and KBEmacs. This section presents several Ada cliches and contrasts them with their Lisp counterparts. (Appendix A presents the complete Ada cliche library.)

Lisp cliches and Ada cliches differ in analogy with the way that the languages Lisp and Ada differ. The primary difference is merely syntactic. Many corresponding cliches in the two languages are, apart from syntax, identical. However, there are a number of semantic differences between the languages, and between their run time environments, which can lead to significant differences between corresponding cliches.

A Simple Cliche

Consider the simple cliche squaring (reproduced below) which was used as an example in the last chapter.

```
(DEFINE-CLICHE SQUARING
    (PRIMARY-ROLES (NUMBER)
     DESCRIBED-ROLES (NUMBER)
     COMMENT "computes the square of {the number}")
  (^ {the input number} 2))
```

This is an example of a cliche which is semantically identical in Ada. If expressed in Ada syntax, the cliche would appear as follows. (This report uses the standard Ada orthographic convention of rendering all, and only, reserved words in lower case.)

```
cliche SQUARING is
   primary roles NUMBER;
   described roles NUMBER;
   comment "computes the square of {the number}";

begin
   return {the input number}**2;
end SQUARING;
```

In order to support the definition of cliches, Ada syntax is extended in two ways. First, `{...}` annotation is supported in analogy with the way it is supported in Lisp. This annotation can either appear as a simple quantity (e.g., `{the input number}`) or in the form of a function call (e.g., `{the operation}(DATA)`).

Second, a new defining form "`cliche`" is introduced in analogy with an Ada procedure definition. As can be seen in the example above, this new form is simply a transliteration of the Lisp form `DEFINE-CLICHE` into an Ada-like syntax. The body is the same as the body of an Ada procedure definition.

The Cliche File_Enumeration

As a more interesting example, consider the cliche file_enumeration (shown on the next page). (In Ada, identifiers cannot contain the character "-".) The cliche file_enumeration reads the records in a file, enumerating them sequentially. (The Annotation "{}" is used to represent things which must be present for syntactic completeness, but which are not important enough to be given names as explicit roles — e.g., the data types of the variables `FILE` and `DATA_RECORD`.)

Like the cliche list-enumeration (discussed in the last chapter), the cliche file_enumeration is a member of the general class of cliches known as enumerators. As such, it has the same basic structure and the same basic roles. The input role is the file to be enumerated. The empty_test (`END_OF_FILE`) tests to see whether all of the records in the file have been read. The procedure `READ` corresponds to both the element_accessor and the step. (Stepping occurs as a side-effect of the procedure `READ`. In addition to reading a record from a file and returning it in the variable which is its second argument (here `DATA_RECORD`), the procedure `READ` moves from one record to the next.) The output role data_record is the series of records enumerated. (The name record cannot be used because it is a reserved word in Ada).

In addition to enumerating the records in the file, the cliche file_enumeration takes care of opening and closing the file. The exception handlers at the end of the begin block specify what to do if some interrupt occurs during the execution of the block. They specify that no matter what happens, the file should be closed. (This is equivalent to the support provided by the Lisp function `WITH-OPEN-FILE`.) In addition, the exception handlers specify that if the exception is caused accessing a file then an error message should be printed and the exception suppressed; otherwise, the exception is passed on to be handled at a higher level.

As discussed in the last chapter, an important aspect of cliches is that they must be designed to fit together. In this chapter it is assumed that a programming organization exists which is producing a number of programs

which interact with a data base represented as a group of files. It is further assumed that a set of conventions has been established for how these programs should be written. Three of these conventions are evident in the cliche file_enumeration.

```
cliche FILE_ENUMERATION is
   primary roles FILE;
   described roles FILE;
   comment "enumerates the records in {the file}";
   constraints
      RENAME("DATA_RECORD", SINGULAR_FORM({the file}));
      DEFAULT({the file_name}, CORRESPONDING_FILE_NAME({the file}));
   end constraints;

   FILE: {};
   DATA_RECORD: {};
begin
   FILE := {the input file};
   OPEN(FILE, IN_FILE, {the file_name});
   while not {END_OF_FILE, the empty_test}(FILE) loop
      {{READ, the element_accessor}, the step}(FILE, DATA_RECORD);
      {DATA_RECORD, the output data_record};
   end loop;
   CLOSE(FILE);
exception
   when DEVICE_ERROR | END_ERROR | NAME_ERROR | STATUS_ERROR =>
      CLOSE(FILE); PUT("Data Base Inconsistent");
   when others => CLOSE(FILE); raise;
end FILE_ENUMERATION;
```

The strong typing in Ada requires that there be different I/O functions for each type of file. Here it is assumed that overloading will be used so that these functions can always be referred to by their standard names (e.g. OPEN, CLOSE, READ, etc.).

Another convention revolves around the handling of exceptions. It is assumed that each program written will be complete in that it will explicitly check for every kind of abnormal condition which could occur. However, programs are allowed to assume that the data base files are in a consistent state. For example, while the cliche file_enumeration explicitly checks for the end of the file being enumerated, it does not check that the file actually exists before opening it. Rather, it is assumed that the cliche will only be used to operate on files which are known to exist when the data base is consistent. In order to deal with the fact that in extraordinary circumstances the data base might not be in a consistent state, a single blanket exception handler (shown above) is included as part of the outermost block of every program that accesses any files.

The first constraint specified as part of the cliche file_enumeration suggests a name to use for the variable holding the records being enumerated. The name chosen reflects a convention governing the way files are named. Suppose that a file contains addresses. It is assumed that the variable containing the file object will be given the plural name ADDRESSES and that if a variable is needed to hold a record from the file it will be given the singular name ADDRESS. (This convention is not a hard and fast one, but rather a stylistic suggestion.)

The constraint RENAME is essentially textual in nature. It replaces all instances of its first parameter with its second parameter. For example, if the file were specified to be the variable ADDRESSES then the identifier DATA_RECORD would be changed to ADDRESS. In addition to whole identifiers, parts of identifiers are also changed — if there were an identifier DATA_RECORD_KEY it would be changed to ADDRESS_KEY. It should be noted that renaming is convenient, but not a fundamental issue. It merely makes it easier for KBEmacs to generate aesthetic program code.

The second constraint specified as part of the cliche file_enumeration specifies a default value for the

file_name role. The function CORRESPONDING_FILE_NAME determines the appropriate external name to use when opening the file. It will be discussed further below.

A Data Structure Cliche

The cliches exhibited so far have all been control structure cliches. However, cliched data structures are just as important as cliched computations. The cliche chain_file_definition captures the stereotyped aspects of the definition of a *chain file*. As used here, the term "chain file" refers to a file which contains records that are chained together in lists. The individual lists are pointed to by records in some other file as shown in Figure 6. The figure shows a file of student records which point to chains of records corresponding to the various classes each student has taken. Note that the index value 0 is used to represent the end of a list.

main file

STUDENT	YEAR	CLASSES
H.B. Bonner	88	1
R.W. Hawk	89	0
T.E. Johnson	88	3

chain file

IDX	COURSE	DATE	GRADE	NEXT
1	Physics 101	Fall 84	A	2
2	English 204	Fall 84	C	0
3	Physics 101	Fall 84	C	4
4	English 204	Fall 84	C	0

Figure 6: An example of a chain file.

The cliche chain_file_definition (shown on the next page) has two roles, the external name of the file being defined and the data fields in the chain record type. The body of the cliche has five parts. A variable is defined which holds a string containing the external name. A type is defined for index values pointing to the chain records. The type of the chain records themselves is then defined. The only predefined field in this record type is the NEXT field which is used to point from one chain record in a list to the next. A package is created which defines the I/O functions which can operate on the chain file being defined. Lastly, a variable which can contain an instance of the chain file is defined. A renaming constraint is used to give aesthetic names to the identifiers in the cliche. The function FILE_NAME_ROOT computes a reasonable name to use based on the external name for the file.

A key feature of the cliche CHAIN_FILE_DEFINITION is that it relies on the generic package CHAINED_IO. (This generic package is defined as part of the package FUNCTIONS. The package FUNCTIONS contains the definition of a number of supporting functions used in the scenario below. The package is presented in Appendix B)

The use of the generic package CHAINED_IO is another example of the fact that KBEmacs tries to take advantage of all preexisting mechanisms. If it were possible, everything about the definition of a chain file would have been relegated to the generic package. Unfortunately this is not possible because, in Ada, the fields of a record are not allowed to be generic parameters.

```
with FUNCTIONS;
use FUNCTIONS;
cliche CHAIN_FILE_DEFINITION is
   primary roles FILE_NAME;
   described roles FILE_NAME;
   comment "defines a file named {the file_name} of chain records";
   constraints
      RENAME("DATA_RECORD", FILE_NAME_ROOT({the file_name}));
   end constraints;

   DATA_RECORDS_NAME: constant STRING := {the file_name};
   subtype DATA_RECORD_INDEX_TYPE is INDEX_TYPE;
   type DATA_RECORD_TYPE is
      record
         {the data};
         NEXT: DATA_RECORD_INDEX_TYPE;
      end record;
   package DATA_RECORD_IO is
      new CHAINED_IO(DATA_RECORD_TYPE, DATA_RECORD_INDEX_TYPE);
   DATA_RECORDS: DATA_RECORD_IO.FILE_TYPE;
end CHAIN_FILE_DEFINITION;
```

A Suite of Cliches Operating on Files

Cliches tend to be defined in tightly knit groups, or suites, which are intended to be used together. For example, this was true of the cliches simple-report, print-out, and tabularized-print-out used in the scenario in the last chapter. In this chapter, a number of cliches are used which revolve around the concept of a file of data records. There are cliches for defining a file (e.g., chain_file_definition) and cliches for operating on a file (e.g., file_enumeration).

An interesting aspect of the suite of cliches which operate on files is that, in addition to passively embodying a number of shared conventions, the cliches have constraints which actively share information. In particular, constraints in an Ada cliche which operates on a file often utilize information obtained from the definition of the file. For example, a constraint in the cliche file_enumeration uses the function `CORRESPONDING_FILE_NAME` in order to determine how to refer to the external name of the file being enumerated. This function works by locating and analyzing the definition of the file being enumerated. It depends on the fact that the file was defined in accordance with one of the standard cliches for defining a file (e.g., chain_file_definition).

As a further example of how cliches which operate on files refer to the definition of those files, consider the cliche chain_enumeration (shown below). This cliche specifies how to enumerate a chain of records in a chain file starting from a record in a main file.

```
cliche CHAIN_ENUMERATION is
   primary roles MAIN_FILE, CHAIN_FILE, MAIN_FILE_KEY;
   described roles MAIN_FILE, CHAIN_FILE, MAIN_FILE_KEY;
   comment "enumerates the chain records in {the chain_file} starting
            from the header record indexed by {the main_file_key}";
   constraints
      RENAME("MAIN_RECORD", SINGULAR_FORM({the main_file}));
      RENAME("CHAIN_RECORD", SINGULAR_FORM({the chain_file}));
      DEFAULT({the main_file_name},
              CORRESPONDING_FILE_NAME({the main_file}));
      DEFAULT({the chain_file_name},
              CORRESPONDING_FILE_NAME({the chain_file}));
      DEFAULT({the main_file_chain_field},
              CHAIN_FIELD({the main_file}, {the chain_file}));
      DEFAULT({the step}, CHAIN_FIELD({the chain_file}, {the chain_file}));
   end constraints;

   CHAIN_FILE: {};
   CHAIN_RECORD: {};
   CHAIN_RECORD_INDEX: {};
   MAIN_FILE: {};
   MAIN_RECORD: {};

   procedure CLEAN_UP is
   begin
      CLOSE(CHAIN_FILE); CLOSE(MAIN_FILE);
   exception
      when STATUS_ERROR => return;
   end CLEAN_UP;
begin
   CHAIN_FILE := {the chain_file};
   MAIN_FILE := {the main_file};
   OPEN(CHAIN_FILE, IN_FILE, {the chain_file_name});
   OPEN(MAIN_FILE, IN_FILE, {the main_file_name});
   READ(MAIN_FILE, MAIN_RECORD, {the input main_file_key});
   CHAIN_RECORD_INDEX := MAIN_RECORD.{the main_file_chain_field};
   while not {NULL_INDEX, the empty_test}(CHAIN_RECORD_INDEX) loop
      {READ, the element_accessor}(CHAIN_FILE, CHAIN_RECORD, CHAIN_RECORD_INDEX);
      {CHAIN_RECORD, the output chain_record};
      CHAIN_RECORD_INDEX := CHAIN_RECORD.{the step};
   end loop;
   CLEAN_UP;
exception
   when DEVICE_ERROR | END_ERROR | NAME_ERROR | STATUS_ERROR =>
      CLEAN_UP; PUT("Data Base Inconsistent");
   when others => CLEAN_UP; raise;
end CHAIN_ENUMERATION;
```

The cliche chain_enumeration is similar to the cliche file_enumeration. In particular, it shares the basic structure of an enumerator and the code associated with opening and closing files. (Note that since two files have to be closed, a helping function (`CLEAN_UP`) is defined which closes both files. This function contains an exception handler which deals with the situation where the files have not both been successfully opened.)

However, the cliche file_enumeration embodies an entirely different method for reading records from a file. Instead of reading records sequentially, the cliche uses explicit indices in order to read successive records in a chain. The input role is the key of a record in a main file which points to a chain of records in a chain file. The empty_test (`NULL_INDEX`) tests whether the end of the chain has been reached. The step steps from one record in the chain to the next by accessing the appropriate record field. The element_accessor (`READ`) reads the appropriate records from the chain file. (When present, the third argument to the procedure `READ` specifies the index/key of the record to read.) The output role chain_record is the series of records in the chain.

The most interesting aspect of the cliche chain_enumeration is the set of constraints. The first four constraints are analogous to the constraints in the cliche file_enumeration. The first two constraints generate mnemonic identifiers based on the names of the files to be operated on. The second two constraints look at the definition of these files in order to determine the appropriate external file name to use when opening the files.

The last two constraints specify how to go from one record in the chain to the next. The function `CHAIN_FIELD` looks at the definitions of two files, and determines what field of the record in the first file contains an index into the second file. The next to last constraint specifies how to get the index of the first record in the chain from the main file record. The last constraint specifies how to step from one chain record to the next.

The Cliche Simple_Report

As a final example of an Ada cliche, consider the cliche simple_report (shown on the next page) in contrast to the Lisp cliche simple-report (reproduced below). These two cliches are a good example of the differences as well as the similarities between Ada cliches and Lisp cliches. The cliches have the same purpose and the same roles. The primary differences between the cliches are purely syntactic.

```
(DEFINE-CLICHE SIMPLE-REPORT
    (PRIMARY-ROLES (ENUMERATOR PRINT-ITEM SUMMARY)
     DESCRIBED-ROLES (FILE-NAME TITLE ENUMERATOR
                      COLUMN-HEADINGS PRINT-ITEM SUMMARY)
     COMMENT "prints a report of {the input structure of the enumerator}"
     CONSTRAINTS
      ((DEFAULT {the file-name} "report.txt")
       (DERIVED {the line-limit}
                (- 65
                   (SIZE-IN-LINES {the print-item})
                   (SIZE-IN-LINES {the summary})))))
  (WITH-OPEN-FILE (REPORT {the file-name} ':OUT)
    (LET* ((DATE (TIME:PRINT-CURRENT-TIME NIL))
           (LINE 66)
           (PAGE 0)
           (TITLE {the title})
           (DATA {the input structure of the enumerator}))
      (FORMAT REPORT "~5%~66: <~A~>~2%~66: <~A~>~%" TITLE DATE)
      (LOOP DO
        (IF ({the empty-test of the enumerator} DATA) (RETURN))
        (WHEN (> LINE {the line-limit})
          (SETQ PAGE (+ PAGE 1))
          (FORMAT REPORT "~/|~%Page:~3D~50: <~A~>~17A~2%" PAGE TITLE DATE)
          (SETQ LINE 3)
          ({the column-headings} {REPORT, modified} {LINE, modified}))
        ({the print-item} {REPORT, modified}
                          {LINE, modified}
                          ({the element-accessor of the enumerator} DATA))
        (SETQ DATA ({the step of the enumerator} DATA)))
      ({the summary} {REPORT, modified}))))
```

However, there are a a number of semantic differences between the cliches. At a relatively trivial level, the textual I/O functions available in Ada are quite different then the ones in Lisp. In particular, there is no construct corresponding to the Lisp function `FORMAT`. Instead, sets of more primitive functions (e.g., `NEW_LINE`, `SET_COL`, `PUT`, etc.) have to be used. At a similar level, the Lisp expression `(TIME:PRINT-CURRENT-TIME NIL)` is replaced by the similar (but not identical) Ada expression `FORMAT_DATE(CLOCK)`. The function `CLOCK` which returns the current date and time is part of the standard Ada package `CALENDAR`. (In Ada calls on zero argument functions are identical in appearance to variable references.) The function `FORMAT_DATE` (defined in the package `FUNCTIONS`) converts a time into a character string. Other examples of minor semantic differences include the way exceptions are handled.

At a more fundamental level, the cliches differ because Ada automatically keeps track of the line number and the page number in an output file. This leads to significant simplifications in the Ada cliche. However, it also leads to a slight complication. In order to trigger a page break after the title is printed, the Ada cliche has to actually print out `60` blank lines in order to force the line counter to be incremented sufficiently. (Note that while the cliche simple-report follows the standard Lisp style of counting things starting with zero, Ada counts lines and pages starting with one.)

A final difference between the two cliches is that the Ada cliche takes advantage of the strong typing enforced by Ada to add two powerful constraints. The function `CORRESPONDING_PRINTING` determines what

should be used to fill in the print_item role based on the type of object which is being enumerated. The function CORRESPONDING_HEADINGS determines what headings should be used based on how the print_item is filled in. As a result, as soon as the type of object being enumerated is known, the print_item and then the column_headings can be automatically filled in.

```
with CALENDAR, FUNCTIONS, TEXT_IO;
use CALENDAR, FUNCTIONS, TEXT_IO;
cliche SIMPLE_REPORT is
   primary roles ENUMERATOR, PRINT_ITEM, SUMMARY;
   described roles FILE_NAME, TITLE, ENUMERATOR, COLUMN_HEADINGS,
                   PRINT_ITEM, SUMMARY;
   comment "prints a report of {the input structure of the enumerator}";
   constraints
      DEFAULT({the file_name}, "report.txt");
      DERIVED({the line_limit},
              66-SIZE_IN_LINES({the print_item})
                -SIZE_IN_LINES({the summary}));
      DEFAULT({the print_item}, CORRESPONDING_PRINTING({the enumerator}));
      DEFAULT({the column_headings},
              CORRESPONDING_HEADINGS({the print_item}));
   end constraints;

   use INT_IO;
   CURRENT_DATE: constant STRING := FORMAT_DATE(CLOCK);
   DATA: {};
   REPORT: TEXT_IO.FILE_TYPE;
   TITLE: STRING(1..{});
   procedure CLEAN_UP is
   begin
      SET_OUTPUT(STANDARD_OUTPUT);
      CLOSE(REPORT);
   exception
      when STATUS_ERROR => return;
   end CLEAN_UP;
begin
   CREATE(REPORT, OUT_FILE, {the file_name});
   DATA := {the input structure of the enumerator};
   SET_OUTPUT(REPORT);
   TITLE := {the title};
   NEW_LINE(4); SET_COL(20); PUT(CURRENT_DATE); NEW_LINE(2);
   SET_COL(13); PUT(TITLE); NEW_LINE(60);
   while not {the empty_test of the enumerator}(DATA) loop
      if LINE > {the line_limit} then
         NEW_PAGE; NEW_LINE; PUT("Page: "); PUT(INTEGER(PAGE-1), 3);
         SET_COL(13); PUT(TITLE);
         SET_COL(61); PUT(CURRENT_DATE); NEW_LINE(2);
         {the column_headings}({CURRENT_OUTPUT, modified});
      end if;
      {the print_item}({CURRENT_OUTPUT, modified},
                       {the element_accessor of the enumerator}(DATA));
      DATA := {the step of the enumerator}(DATA);
   end loop;
   {the summary};
   CLEAN_UP;
exception
   when DEVICE_ERROR | END_ERROR | NAME_ERROR | STATUS_ERROR =>
      CLEAN_UP; PUT("Data Base Inconsistent");
   when others => CLEAN_UP; raise;
end SIMPLE_REPORT;
```

The functions CORRESPONDING_PRINTING and CORRESPONDING_HEADINGS operate in one of two modes.

In general, cliches will have been defined which specify how to print out a given type of object in a report, and how to print the corresponding headings. If this is the case, then the functions `CORRESPONDING_PRINTING` and `CORRESPONDING_HEADINGS` merely retrieve the appropriate cliches. However, if there are no such cliches, then the functions `CORRESPONDING_PRINTING` and `CORRESPONDING_HEADINGS` use a simple program generator in order to construct appropriate code based on the definition of the type of object in question. Both of these modes will be illustrated in the scenario below.

Their differences not withstanding, the cliches simple-report and simple_report are most notable for their similarities. At a high enough level of abstraction, these similarities could be captured in a simple report program cliche which was semantically identical in Lisp and Ada. The cliches simple-report and simple_report are specializations of that abstract cliche. For example, the abstract cliche would require only that some method be employed to keep track of line numbers and page numbers. The cliches simple-report and simple_report differ in the way this is achieved because Lisp and Ada differ in the support they provide.

Limits to the Power of Constraints

The power of a constraint follows directly from the power of the functions used in the constraining expression. Constraining expressions can utilize any of the standard functions and operators in the target language, and the programmer is free to write any additional function he desires. As a result, there is essentially no theoretical limitation to the computation which can be performed by a constraint expression.

However, as a practical matter, the power of constraints is limited by the fact that writing new functions to use in constraints is not particularly easy. A major problem is that instances of `{...}` annotation in constraint expressions evaluate to direct pointers into the internal knowledge representation used by KBEmacs. This allows for great flexibility in what can be done by a constraint function, but requires that the writer of a constraint function be conversant in this internal representation. In contrast, when defining the body of a cliche the programmer need not understand anything other than the target language and `{...}` annotation.

It is expected that the typical programmer will not write new functions to use in constraints, but rather will only write expressions which use predefined functions. As a result, the power of constraints is in effect limited by the constraint functions provided by the designer of the basic cliche library.

Viewed from this perspective, the constraints in the cliches above fall into three categories. It is expected that simple constraints such as the one deriving the line_limit role in the cliche simple_report will be common. They depend only on standard functions and a few general purpose constraint functions such as `SIZE_IN_LINES`. There is no reason why such constraints could not be included as part of idiosyncratic cliches defined by programmers.

The constraint functions `CORRESPONDING_FILE_NAME` and `CHAIN_FIELD` are at an intermediate level of complexity. It is plausible that such functions would be defined as part of defining a suite of cliches. It is even plausible that a programmer might use these functions himself if he defined a new cliche to add to the suite. However, the constraint functions are of no use outside of the suite, and it is not plausible that the typical programmer would define additional functions of this nature.

Finally, the inclusion of constraint functions such as `CORRESPONDING_PRINTING` and `CORRESPONDING_HEADINGS` would seem to be an unusual event. Writing such a function not only requires a knowledge of the internal knowledge representation used by KBEmacs, but is difficult in its own right. They are included in the cliches in this scenario in order to demonstrate the full power of constraints and to illustrate the way constraints can be used as an interface between KBEmacs and other programming tools — e.g., in this case a program generator.

Defining Data Structures

In the first part of the scenario below, the programmer constructs an Ada package which defines a data base. In the second and third parts of the scenario, the programmer constructs two report programs which operate on this data base. Before beginning the definition of the data base package, it is useful to set the scene by describing the structure of the data base.

The Data Base Used in the Scenario

The data base contains information about repairs performed on instruments sold by a imaginary marketing company. The company sells and maintains these instruments but does not build them. As illustrated in Figure 7, the data base is composed of four files: a file specifying the name and maker of each model of instrument the company sells, a file specifying the kinds of defects which can occur on each model, a file describing each individual unit sold, and a file describing each individual repair performed on any of these units. In addition to illustrating the structure of the data base, Figure 7 shows an example of the kind of data that might be placed in the data base.

defects file

KEY	NAME	MODEL
OS-03	Power supply thermistor blown	OS1
GA-11	Control board cold solder joint	GA2
GA-32	Clogged gas injection port	GA2

models file

KEY	NAME	MAKER
OS1	Opal Sorter	Perth Mining
GA2	Gas Analyzer	Benson Labs

units file

KEY	MODEL	REPAIR
OS1-271	OS1	3
GA2-342	GA2	4

repairs file

IDX	DATE	DEFECT	COMMENT	NEXT
1	9/14/83	GA-32	Probably caused by humidity.	0
2	1/23/85	GA-11	Took two days to find.	1
3	2/25/85	OS-03	Sorter arm got stuck.	0
4	3/19/85	GA-32	Port Diameter seems below specs.	2

Figure 7: The example data base.

In the interest of brevity, the data base has been simplified to its barest essentials. A real data base would probably contain more files, and would certainly have many more fields in each record. However, these simplifications do not alter the basic nature of the scenario. Adding more files or fields would make the programs constructed in the scenario bigger, but it would not make them fundamentally any more complex.

Directions for a Human Assistant

The Ada package to be constructed defines the four data base files discussed above. The following set of directions might be used to tell a person how to implement the package.

```
Define a package MAINTENANCE_FILES which defines four files --
 A keyed file "models.data" with a 3 character key and the fields:
   NAME: STRING(1..16); MAKER: STRING(1..16).
 A keyed file "defects.data" with a 5 character key and the fields:
   NAME: STRING(1..32); MODEL: key to the file of models.
 A chain file "repairs.data" with the fields:
   DATE: TIME; DEFECT: key to the file of defects; COMMENT: STRING(1..32).
   Each chain should be ordered by date with the most recent record first.
 A keyed file "units.data" with a 7 character key and the fields:
   MODEL: key to the file of models; REPAIR: index into the file of repairs.
```

The above directions are somewhat different in tone from the directions in the last chapter. Although they make use of high level terms such as "keyed file" and "chain file", they consist mostly of detailed specifications for the records in the file. This level of detail would seem unavoidable, since this is a situation where the details are important.

Switching to Ada Mode

The scenario in this chapter is illustrated with screen images of exactly the same form as the ones in the last chapter. The first thing the programmer does in the scenario (see Screen 26) is to tell KBEmacs to use the language Ada (as opposed to Lisp which is the default). He does this by using the knowledge-based command "`Use language`".

The Lisp Machine editor was extended by adding an Ada mode. As will be seen below, this mode enables several of the program oriented Emacs commands (e.g., `m-.` (goto definition) and `c-shift-C` (compile definition)) to operate properly on Ada programs.

In order to shorten the scenario it is assumed that the programmer has already defined the files `MODELS` and `DEFECTS`. In Screen 26, the programmer reads in a file containing the partially implemented package.

```
s-X Use language ADA. <end>
c-X c-F Find file <KBE.DEMO>MAINTENANCE_FILES.ADA
```

cursor motion

```
with FUNCTIONS;
use FUNCTIONS;
package MAINTENANCE_FILES is
   MODELS_NAME: constant STRING := "models.data";
   type MODEL_KEY_TYPE is STRING(1..3);
   type MODEL_TYPE is
      record
         NAME: STRING(1..16);
         MAKER: STRING(1..16);
      end record;
   package MODEL_IO is new KEYED_IO(MODEL_TYPE, MODEL_KEY_TYPE);
   MODELS: MODEL_IO.FILE_TYPE;

   DEFECTS_NAME: constant STRING := "defects.data";
   type DEFECT_KEY_TYPE is STRING(1..5);
   type DEFECT_TYPE is
      record
         NAME: STRING(1..32);
         MODEL: MODEL_KEY_TYPE;
      end record;
   package DEFECT_IO is new KEYED_IO(DEFECT_TYPE, DEFECT_KEY_TYPE);
   DEFECTS: DEFECT_IO.FILE_TYPE;
   

end MAINTENANCE_FILES;
```

```
KBEmacs (ADA) <KBE.DEMO>MAINTENANCE_FILES.ADA
```

Defining a File

In Screen 27, the programmer begins to define the file REPAIRS. He does this by moving to the end of the package (see Screen 26) and then inserting an instance of the cliche chain_file_definition discussed above. Note the way renaming has been applied to the various identifiers in the cliche.

s-X Insert a chain_file_definition of "repairs.data". <end>

```
with FUNCTIONS;
use FUNCTIONS;
package MAINTENANCE_FILES is
   MODELS_NAME: constant STRING := "models.data";
   type MODEL_KEY_TYPE is STRING(1..3);
   type MODEL_TYPE is
      record
         NAME: STRING(1..16);
         MAKER: STRING(1..16);
      end record;
   package MODEL_IO is new KEYED_IO(MODEL_TYPE, MODEL_KEY_TYPE);
   MODELS: MODEL_IO.FILE_TYPE;

   DEFECTS_NAME: constant STRING := "defects.data";
   type DEFECT_KEY_TYPE is STRING(1..5);
   type DEFECT_TYPE is
      record
         NAME: STRING(1..32);
         MODEL: MODEL_KEY_TYPE;
      end record;
   package DEFECT_IO is new KEYED_IO(DEFECT_TYPE, DEFECT_KEY_TYPE);
   DEFECTS: DEFECT_IO.FILE_TYPE;

   REPAIRS_NAME: constant STRING := "repairs.data";
   subtype REPAIR_INDEX_TYPE is INDEX_TYPE;
   type REPAIR_TYPE is
      record
         {the data};
         NEXT: REPAIR_INDEX_TYPE;
      end record;
   package REPAIR_IO is new CHAINED_IO(REPAIR_TYPE, REPAIR_INDEX_TYPE);
   REPAIRS: REPAIR_IO.FILE_TYPE;
end MAINTENANCE_FILES;
```

KBEmacs (ADA) <KBE.DEMO>MAINTENANCE_FILES.ADA

Direct Editing

In Screen 28, the programmer moves to the data role and uses direct editing to specify the fields of a repair record. The most important thing to notice about this screen (and about the last screen) is that KBEmacs can assist in the construction of a data structure definition in exactly the same way that it can assist in the construction of a program.

s-N *direct editing*

```
with FUNCTIONS;
use FUNCTIONS;
package MAINTENANCE_FILES is
   MODELS_NAME: constant STRING := "models.data";
   type MODEL_KEY_TYPE is STRING(1..3);
   type MODEL_TYPE is
      record
         NAME: STRING(1..16);
         MAKER: STRING(1..16);
      end record;
   package MODEL_IO is new KEYED_IO(MODEL_TYPE, MODEL_KEY_TYPE);
   MODELS: MODEL_IO.FILE_TYPE;

   DEFECTS_NAME: constant STRING := "defects.data";
   type DEFECT_KEY_TYPE is STRING(1..5);
   type DEFECT_TYPE is
      record
         NAME: STRING(1..32);
         MODEL: MODEL_KEY_TYPE;
      end record;
   package DEFECT_IO is new KEYED_IO(DEFECT_TYPE, DEFECT_KEY_TYPE);
   DEFECTS: DEFECT_IO.FILE_TYPE;

   REPAIRS_NAME: constant STRING := "repairs.data";
   subtype REPAIR_INDEX_TYPE is INDEX_TYPE;
   type REPAIR_TYPE is
      record
         DATE: TIME;
         DEFECT: DEFECT_KEY_TYPE;
         COMMENT: STRING(1..32);
         NEXT: REPAIR_INDEX_TYPE;
      end record;
   package REPAIR_IO is new CHAINED_IO(REPAIR_TYPE, REPAIR_INDEX_TYPE);
   REPAIRS: REPAIR_IO.FILE_TYPE;
end MAINTENANCE_FILES;
```

KBEmacs (ADA) <KBE.DEMO>MAINTENANCE_FILES.ADA

Screen 28

Analyzing Direct Editing Done by the Programmer

Having finished the definition of the file REPAIRS, the programmer tells KBEmacs that, for the moment, he is done with the package MAINTENANCE_FILES (see Screen 29). KBEmacs then proceeds to analyze the editing done by the programmer.

Taking Care of Details

A fundamental underpinning of Ada is the notion of a *package* which represents a group of functions or data structures or both. In general, related entities are defined together in a package. For example, here the files in the date base are being defined in a package named MAINTENANCE_FILES.

The names of the objects defined in a package are local to that package. In order to refer to these objects from another package one must do two things. First, one must specify that the second package is to be compiled with the first package. (The first line of Screen 29 specifies that the package MAINTENANCE_FILES is to be compiled with the packages CALENDAR and FUNCTIONS.) Second, one must use the correct name for the object being referred to. The basic way to do this is to use a compound name of the form *package_name.object_name*. For example, the second to last line of Screen 29 refers to the data type FILE_TYPE defined in the package REPAIR_IO as REPAIR_IO.FILE_TYPE. Alternatively, one can specify that the second package uses the first package (see the second line of Screen 29) and then refer to the object by its simple name if this name is not ambiguous. For example, the third to last line of Screen 29 refers to the generic package CHAINED_IO (which is defined in the package FUNCTIONS) directly.

The above digression into the details of Ada is necessary in order to be able to understand an error made by the programmer in Screen 28. The programmer specified that the DATE field of a repair record had the data type TIME. Assumedly, he intended to refer to the data type CALENDAR.TIME which is defined in the standard Ada package CALENDAR. However, he did not use the proper name to refer to it. (This sloppy reference is already present in the directions hypothesized above.)

When KBEmacs analyzes the direct editing in Screen 28, it notices that the unknown data type TIME is being referred to. KBEmacs then reasons that since the only TIME data type defined anywhere is in the CALENDAR package, this must be what the programmer is trying to refer to. In order to fix things up, KBEmacs adds the package CALENDAR to the "with" and "use" clauses of the package MAINTENANCE_FILES.

This is a small example of an important ability KBEmacs attempts to demonstrate — taking care of details. This is particularly useful when operating on Ada, because Ada code contains a significant amount of redundancy which can be quite irritating to specify. (Note that this redundancy is intended primarily for the benefit of readers and maintainers, not implementors.) KBEmacs attempts to automatically derive redundant information whenever possible so that programmers do not have to specify it explicitly.

Recognizing a File Definition

When analyzing the package in Screen 28, KBEmacs recognizes that a new data file (REPAIRS) is being defined. This recognition is done by a special purpose module of KBEmacs which notes the existence of the definition for later reference. In addition to noting the existence of the definition, KBEmacs automatically defines cliches which specify how to print out the records in the file. In the second line of the command area of Screen 29, KBEmacs informs the programmer that two cliches have been defined: print_repair which prints out a repair record, and print_repair_headings which prints out the appropriate column headings. The names of these cliches are stored as part of the information about the file so that they can be retrieved by the functions CORRESPONDING_PRINTING and CORRESPONDING_HEADINGS (used as constraints in the cliche simple_report discussed above).

s-<end> ***Finish editing the package MAINTENANCE_FILES. <end>***
The cliches PRINT_REPAIR_HEADINGS and PRINT_REPAIR defined.

```
with CALENDAR, FUNCTIONS;
use CALENDAR, FUNCTIONS;
package MAINTENANCE_FILES is
   MODELS_NAME: constant STRING := "models.data";
   type MODEL_KEY_TYPE is STRING(1..3);
   type MODEL_TYPE is
      record
         NAME: STRING(1..16);
         MAKER: STRING(1..16);
      end record;
   package MODEL_IO is new KEYED_IO(MODEL_TYPE, MODEL_KEY_TYPE);
   MODELS: MODEL_IO.FILE_TYPE;

   DEFECTS_NAME: constant STRING := "defects.data";
   type DEFECT_KEY_TYPE is STRING(1..5);
   type DEFECT_TYPE is
      record
         NAME: STRING(1..32);
         MODEL: MODEL_KEY_TYPE;
      end record;
   package DEFECT_IO is new KEYED_IO(DEFECT_TYPE, DEFECT_KEY_TYPE);
   DEFECTS: DEFECT_IO.FILE_TYPE;

   REPAIRS_NAME: constant STRING := "repairs.data";
   subtype REPAIR_INDEX_TYPE is INDEX_TYPE;
   type REPAIR_TYPE is
      record
         DATE: TIME;
         DEFECT: DEFECT_KEY_TYPE;
         COMMENT: STRING(1..32);
         NEXT: REPAIR_INDEX_TYPE;
      end record;
   package REPAIR_IO is new CHAINED_IO(REPAIR_TYPE, REPAIR_INDEX_TYPE);
   REPAIRS: REPAIR_IO.FILE_TYPE;
end MAINTENANCE_FILES;
```

KBEmacs (ADA) <KBE.DEMO>MAINTENANCE_FILES.ADA

Automatically Defined Cliches

In Screen 30, the programmer uses the standard editing command `m-.` (move to definition) in order to examine the cliches defined by KBEmacs. These cliches have been added to the file containing the library of Ada cliches.

The cliches are created by the same special purpose program generator used by the functions `CORRESPONDING_PRINTING` and `CORRESPONDING_HEADINGS`. In order to create code for printing a record, the generator calls itself recursively on each field of the record and combines the chunks of code produced. Each field is printed in a standardized way based on its data type. For example, any field whose data type is `CALENDAR.TIME` is printed out using the function `FORMAT_DATE`.

Actual cliche definitions are created (as opposed to simply generating the corresponding code at the moment the user desires to print a repair) in order to allow the programmer to modify the cliches. This would be desirable even if the program generator were a great deal better than the one used, because people differ widely in what they consider to be the aesthetic way to print something.

m-. *Edit definition of* PRINT_REPAIR_HEADINGS

```
with TEXT_IO;
use TEXT_IO;
cliche PRINT_REPAIR_HEADINGS is
   comment "prints headings for printing repair records";

begin
   PUT("   Date     Defect                 Comment");
   NEW_LINE(2);
end PRINT_REPAIR_HEADINGS;


with FUNCTIONS, TEXT_IO;
use FUNCTIONS, TEXT_IO;
cliche PRINT_REPAIR is
   primary roles REPAIR;
   described roles REPAIR;
   comment "prints out the fields of {the repair}";

   REPAIR: REPAIR_TYPE;
begin
   REPAIR := {the input repair};
   PUT(FORMAT_DATE(REPAIR.DATE)); SET_COL(13); PUT(REPAIR.DEFECT);
   SET_COL(20); PUT(REPAIR.COMMENT); NEW_LINE;
end PRINT_REPAIR;
```

KBEmacs (ADA) <KBE.DEMO>CLICHE-LIBRARY.ADA

The Cliche Read

In Screen 31, the programmer modifies the cliche print_repair so that, in addition to printing out the key of the defect repaired, it reads the corresponding defect record and prints out the full name of the defect. The cliche print_repair_headings is modified analogously.

In order to specify how the cliche print_repair is to be modified, the programmer makes an inline reference to the cliche read (see Appendix A). This cliche has two roles: the file to be read and the key of the desired record. This is a particularly nice example of the use of an embedded cliche reference for two reasons. First, notice that "`.NAME`" is used after the cliche annotation in order to select the `NAME` field of the record read just as if the annotation were a variable reference. Second, the annotation makes it possible to specify a read operation as if it were a function call even though the read operation is a procedure call. This enables the desired computation to be represented as a simple expression.

direct editing

```
with TEXT_IO;
use TEXT_IO;
cliche PRINT_REPAIR_HEADINGS is
   comment "prints headings for printing repair records";

begin
  PUT("   Date     Defect   Description/Comment"); NEW_LINE(2);
end PRINT_REPAIR_HEADINGS;


with FUNCTIONS, TEXT_IO;
use FUNCTIONS, TEXT_IO;
cliche PRINT_REPAIR is
   primary roles REPAIR;
   described roles REPAIR;
   comment "prints out the fields of {the repair}";

   REPAIR: REPAIR_TYPE;
begin
   REPAIR := {the input repair};
   PUT(FORMAT_DATE(REPAIR.DATE)); SET_COL(13); PUT(REPAIR.DEFECT);
   SET_COL(20); PUT({a read of DEFECTS and (REPAIR.DEFECT)}.NAME);
   NEW_LINE; SET_COL(22); PUT(REPAIR.COMMENT); NEW_LINE;
end PRINT_REPAIR;
```

KBEmacs (ADA) <KBE.DEMO>CLICHE-LIBRARY.ADA

Screen 31

Analyzing Direct Editing Done by the Programmer

Having finished the cliches print_repair and print_repair_headings, the programmer tells KBEmacs that he is done with each of them. In addition, he compiles the cliches so that they are available for use (see Screen 32).

When it analyzes the editing done by the programmer in Screen 31, KBEmacs replaces the embedded reference to the cliche read with the corresponding Ada code. As can be seen in Screen 32, this leads to a surprisingly large number of changes to the cliche print_repair. First of all, the package `MAINTENANCE_FILES` is added to the "`with`" and "`use`" clauses in order to render the reference to the identifier `DEFECTS` meaningful. In addition, a "`use`" clause for the package `DEFECT_IO` is added to the declarations in order to make available the appropriate I/O procedures for the file `DEFECTS`. Next, KBEmacs adds a call on the procedure `READ`, a variable `DEFECT` to hold the record read, code which opens and closes the file `DEFECTS`, and code which handles any error conditions which might arise.

```
s-<end> Finish editing the cliche PRINT_REPAIR. <end>
c-shift-C Compiling PRINT_REPAIR
cursor motion
s-<end> Finish editing the cliche PRINT_REPAIR_HEADINGS. <end>
c-shift-C Compiling PRINT_REPAIR_HEADINGS
```

```
with TEXT_IO;
use TEXT_IO;
cliche PRINT_REPAIR_HEADINGS is
   comment "prints headings for printing repair records";

begin
  PUT("   Date     Defect   Description/Comment"); NEW_LINE(2);
end PRINT_REPAIR_HEADINGS;


with FUNCTIONS, MAINTENANCE_FILES, TEXT_IO;
use FUNCTIONS, MAINTENANCE_FILES, TEXT_IO;
cliche PRINT_REPAIR is
   primary roles REPAIR;
   described roles REPAIR;
   comment "prints out the fields of {the repair}";

   use DEFECT_IO;
   DEFECT: DEFECT_TYPE;
   REPAIR: REPAIR_TYPE;
begin
   OPEN(DEFECTS, IN_FILE, DEFECTS_NAME);
   REPAIR := {the input repair};
   READ(DEFECTS, DEFECT, REPAIR.DEFECT);
   PUT(FORMAT_DATE(REPAIR.DATE)); SET_COL(13); PUT(REPAIR.DEFECT);
   SET_COL(20); PUT(DEFECT.NAME); NEW_LINE;
   SET_COL(22); PUT(REPAIR.COMMENT); NEW_LINE;
   CLOSE(DEFECTS);
exception
   when DEVICE_ERROR | END_ERROR | NAME_ERROR | STATUS_ERROR =>
      CLOSE(DEFECTS); PUT("Data Base Inconsistent");
   when others => CLOSE(DEFECTS); raise;
end PRINT_REPAIR;
```

Defining the File UNITS

In Screen 33, the programmer finishes the definition of the package MAINTENANCE_FILES by defining the file UNITS. This is done by using the cliche keyed_file_definition (see Appendix A). This cliche is very similar to the cliche chain_file_definition. In the interest of brevity, the process of defining the file UNITS is compressed into a single screen since it is closely analogous to the process (discussed in detail above) of defining the file REPAIRS. The only real difference is that the programmer must specify the length of the key (here 7) as well as the fields in the file.

Evaluating the Commands Used

The following summary shows all of the KBEmacs commands which are necessary in order to define the package MAINTENANCE_FILES.

```
Insert a keyed_file_definition of "models.data".
direct editing of MODELS file record type and key length.
Insert a keyed_file_definition of "defects.data".
direct editing of DEFECTS file record type and key length.
Insert a chain_file_definition of "repairs.data".
direct editing of REPAIRS file record type.
Insert a keyed_file_definition of "units.data".
direct editing of UNITS file record type and key length.
```

A comparison of this summary with the directions (reproduced below) hypothesized at the start of this section shows two interesting things. First, since the direct editing corresponds to the detailed specification of the record types and key lengths, it can be seen that the two sets of directions are almost identical.

```
Define a package MAINTENANCE_FILES which defines four files --
 A keyed file "models.data" with a 3 character key and the fields:
   NAME: STRING(1..16); MAKER: STRING(1..16).
 A keyed file "defects.data" with a 5 character key and the fields:
   NAME: STRING(1..32); MODEL: key to the file of models.
 A chain file "repairs.data" with the fields:
   DATE: TIME; DEFECT: key to the file of defects; COMMENT: STRING(1..32).
   Each chain should be ordered by date with the most recent record first.
 A keyed file "units.data" with a 7 character key and the fields:
   MODEL: key to the file of models; REPAIR: index into the file of repairs.
```

Second, there is nothing in the KBEmacs commands which corresponds to the admonition that the repair records in a chain should be ordered by DATE with the most recent first. It should be noted that there is nothing in the package MAINTENANCE_FILES which corresponds to this either (see Screen 33). The fact that there is no effective way to state this restriction as part of the file definition could be looked at as a defect of Ada. It is certainly a defect of KBEmacs that the system currently has no way to represent such information.

m-. ***Edit definition of*** MAINTENANCE_FILES
s-X Insert a keyed_file_definition of "units.data". <end>
direct editing
s-<end> ***Finish editing the package MAINTENANCE_FILES. <end>***
The cliches PRINT_UNIT_HEADINGS and PRINT_UNIT defined.

```
with CALENDAR, FUNCTIONS;
use CALENDAR, FUNCTIONS;
package MAINTENANCE_FILES is
   MODELS_NAME: constant STRING := "models.data";
   type MODEL_KEY_TYPE is STRING(1..3);
   type MODEL_TYPE is
      record
         NAME: STRING(1..16);
         MAKER: STRING(1..16);
      end record;
   package MODEL_IO is new KEYED_IO(MODEL_TYPE, MODEL_KEY_TYPE);
   MODELS: MODEL_IO.FILE_TYPE;

   DEFECTS_NAME: constant STRING := "defects.data";
   type DEFECT_KEY_TYPE is STRING(1..5);
   type DEFECT_TYPE is
      record
         NAME: STRING(1..32);
         MODEL: MODEL_KEY_TYPE;
      end record;
   package DEFECT_IO is new KEYED_IO(DEFECT_TYPE, DEFECT_KEY_TYPE);
   DEFECTS: DEFECT_IO.FILE_TYPE;

   REPAIRS_NAME: constant STRING := "repairs.data";
   subtype REPAIR_INDEX_TYPE is INDEX_TYPE;
   type REPAIR_TYPE is
      record
         DATE: TIME;
         DEFECT: DEFECT_KEY_TYPE;
         COMMENT: STRING(1..32);
         NEXT: REPAIR_INDEX_TYPE;
      end record;
   package REPAIR_IO is new CHAINED_IO(REPAIR_TYPE, REPAIR_INDEX_TYPE);
   REPAIRS: REPAIR_IO.FILE_TYPE;

   UNITS_NAME: constant STRING := "units.data";
   type UNIT_KEY_TYPE is STRING(1..7);
   type UNIT_TYPE is
      record
         MODEL: MODEL_KEY_TYPE;
         REPAIR: REPAIR_INDEX_TYPE;
      end record;
   package UNIT_IO is new KEYED_IO(UNIT_TYPE, UNIT_KEY_TYPE);
   UNITS: UNIT_IO.FILE_TYPE;
end MAINTENANCE_FILES;
```

KBEmacs (ADA) <KBE.DEMO>MAINTENANCE_FILE.ADA

Screen 33

Constraint Propagation

In this part of the scenario, the programmer constructs a program UNIT_REPAIR_REPORT which prints out a report of all the repairs which have been performed on a given unit. The process of constructing this program is quite similar to the process of constructing the program REPORT-TIMINGS shown in the last chapter. The most significant difference (besides the fact that one program is written in Lisp and the other is written in Ada) is that a large portion of the program UNIT_REPAIR_REPORT is filled in by means of various constraints.

Directions for a Human Assistant

The following directions might be given to a person who was asked to write the program UNIT_REPAIR_REPORT.

```
Define a simple report program UNIT_REPAIR_REPORT.  Enumerate the chain of
repairs associated with a unit record, printing each one.  Query the user for
the key (UNIT_KEY) of the unit record to start from.  Print the title
("Report of Repairs on Unit " & UNIT_KEY).  Do not print a summary.
```

These directions assume an understanding of the concepts "simple report", "enumerate a chain", and "query the user for a key". They also assume an understanding of the structure of the data base of units and repairs.

Constructing the Program

In Screen 34, the programmer begins the construction of the program UNIT_REPAIR_REPORT by creating a file to hold the program. He then instantiates the cliche simple_report discussed above. This is done with a syntactic variant of the knowledge-based command "Define" which makes it possible to specify the top level structure of the program at the same time that the initial definition is created.

```
c-X c-F Find file <KBE.DEMO>UNIT_REPAIR_REPORT.ADA
(New File)
s-X Define a simple_report procedure UNIT_REPAIR_REPORT. <end>
```

```
with CALENDAR, FUNCTIONS, TEXT_IO;
use CALENDAR, FUNCTIONS, TEXT_IO;
procedure UNIT_REPAIR_REPORT is
   use INT_IO;
   CURRENT_DATE: constant STRING := FORMAT_DATE(CLOCK);
   DATA: {};
   REPORT: TEXT_IO.FILE_TYPE;
   TITLE: STRING(1..{});
   procedure CLEAN_UP is
      begin
         SET_OUTPUT(STANDARD_OUTPUT);
         CLOSE(REPORT);
      exception
         when STATUS_ERROR => return;
      end CLEAN_UP;
begin
   CREATE(REPORT, OUT_FILE, "report.txt");
   TITLE := {the title};
   SET_OUTPUT(REPORT);
   NEW_LINE(4); SET_COL(20); PUT(CURRENT_DATE); NEW_LINE(2);
   SET_COL(13); PUT(TITLE); NEW_LINE(60);
   DATA := {the input structure of the enumerator};
   while not {the empty_test of the enumerator}(DATA) loop
      if LINE > 64 then
         NEW_PAGE; NEW_LINE; PUT("Page: "); PUT(INTEGER(PAGE-1), 3);
         SET_COL(13); PUT(TITLE); SET_COL(61); PUT(CURRENT_DATE); NEW_LINE(2);
         {the column_headings}({CURRENT_OUTPUT, modified});
      end if;
      {the print_item}({CURRENT_OUTPUT, modified},
                       {the element_accessor of the enumerator}(DATA));
      DATA := {the step of the enumerator}(DATA);
   end loop;
   {the summary};
   CLEAN_UP;
exception
   when DEVICE_ERROR | END_ERROR | NAME_ERROR | STATUS_ERROR =>
      CLEAN_UP; PUT("Data Base Inconsistent");
   when others => CLEAN_UP; raise;
end UNIT_REPAIR_REPORT;
```

```
KBEmacs (ADA) <KBE.DEMO>UNIT_REPAIR_REPORT.ADA
```

The Propagation of Design Decisions

The central design decision in any report program is deciding what values are to be reported. When using the cliche simple_report, this decision is specified by selecting an appropriate enumerator. In Screen 35, the programmer specifies that the enumerator should be a chain_enumeration which enumerates a chain of records in the file `REPAIRS` pointed to from a record in the file `UNITS`. Note that several parts of the code produced are specified by the constraints in the cliche chain_enumeration (described above). In particular, the `REPAIR` field is used to get the initial index into the chain file, and the `NEXT` field is used to get the index of each successive repair record. (The cursor in Screen 35 is placed on the read of the file `UNITS` on the theory that changes to the body of a procedure are more interesting than changes to the declarations or to the code which opens files.)

Once the enumerator has been specified, the constraints in the cliche simple_report (described above) fill in most of the rest of the program `UNIT_REPAIR_REPORT`. In particular, the cliche print_repair is used to fill the print_item role and the cliche print_repair_headings is used to fill in the column_headings role. The line_limit is changed from `64` to `63` since the print_repair prints out two lines of output, whereas the default assumption used by the constraint function `SIZE_IN_LINES` in Screen 34 was that the print_out would produce one line of output.

Taking Care of Details

Another thing to note about the code in Screen 35 is that a number of variable declarations and the like have been added to the program. This is another example of how KBEmacs takes care of programming details. It is interesting to note that the data types in these declarations are not specified as part of the cliches used. Rather, KBEmacs computes what data types should be used based on the definitions of the files used, and the specifications for the procedures which operate on the variables.

s-X Fill the enumerator with a chain_enumeration of UNITS and REPAIRS. <end>

```
with CALENDAR, FUNCTIONS, MAINTENANCE_FILES, TEXT_IO;
use CALENDAR, FUNCTIONS, MAINTENANCE_FILES, TEXT_IO;
procedure UNIT_REPAIR_REPORT is
   use DEFECT_IO, REPAIR_IO, UNIT_IO, INT_IO;
   CURRENT_DATE: constant STRING := FORMAT_DATE(CLOCK);
   DEFECT: DEFECT_TYPE;
   REPAIR: REPAIR_TYPE;
   REPAIR_INDEX: REPAIR_INDEX_TYPE;
   REPORT: TEXT_IO.FILE_TYPE;
   TITLE: STRING(1..{});
   UNIT: UNIT_TYPE;
   procedure CLEAN_UP is
      begin
         SET_OUTPUT(STANDARD_OUTPUT);
         CLOSE(DEFECTS); CLOSE(REPAIRS); CLOSE(UNITS); CLOSE(REPORT);
      exception
         when STATUS_ERROR => return;
      end CLEAN_UP;
begin
   OPEN(DEFECTS, IN_FILE, DEFECTS_NAME); OPEN(REPAIRS, IN_FILE, REPAIRS_NAME);
   OPEN(UNITS, IN_FILE, UNITS_NAME); CREATE(REPORT, OUT_FILE, "report.txt");
   TITLE := {the title};
   SET_OUTPUT(REPORT);
   NEW_LINE(4); SET_COL(20); PUT(CURRENT_DATE); NEW_LINE(2);
   SET_COL(13); PUT(TITLE); NEW_LINE(60);
   READ(UNITS, UNIT, {the main_file_key});
   REPAIR_INDEX := UNIT.REPAIR;
   while not NULL_INDEX(REPAIR_INDEX) loop
      READ(REPAIRS, REPAIR, REPAIR_INDEX);
      if LINE > 63 then
         NEW_PAGE; NEW_LINE; PUT("Page: "); PUT(INTEGER(PAGE-1), 3);
         SET_COL(13); PUT(TITLE); SET_COL(61); PUT(CURRENT_DATE); NEW_LINE(2);
         PUT("   Date     Defect   Description/Comment"); NEW_LINE(2);
      end if;
      READ(DEFECTS, DEFECT, REPAIR.DEFECT);
      PUT(FORMAT_DATE(REPAIR.DATE)); SET_COL(13); PUT(REPAIR.DEFECT);
      SET_COL(20); PUT(DEFECT.NAME); NEW_LINE;
      SET_COL(22); PUT(REPAIR.COMMENT); NEW_LINE;
      REPAIR_INDEX := REPAIR.NEXT;
   end loop;
   {the summary};
   CLEAN_UP;
exception
   when DEVICE_ERROR | END_ERROR | NAME_ERROR | STATUS_ERROR =>
      CLEAN_UP; PUT("Data Base Inconsistent");
   when others => CLEAN_UP; raise;
end UNIT_REPAIR_REPORT;
```

KBEmacs (ADA) <KBE.DEMO>UNIT_REPAIR_REPORT.ADA

Screen 35

The Cliche Query_User_For_Key

After specifying the enumerator in Screen 35, the only roles which are left unfilled are the title, the main_file_key, and the summary. In Screen 36, the programmer specifies that the user of the program UNIT_REPAIR_REPORT should be queried in order to determine what main file key to use. This is done with the cliche query_user_for_key (shown below).

```
with TEXT_IO;
use TEXT_IO;
cliche QUERY_USER_FOR_KEY is
   primary roles FILE;
   described roles FILE;
   comment "queries the user for a key to a record in {the file}";
   constraints
      RENAME("DATA_RECORD", SINGULAR_FORM({the file}));
      DEFAULT({the file_name}, CORRESPONDING_FILE_NAME({the file}));
   end constraints;

   DATA_RECORD: {};
   DATA_RECORD_KEY: {};
   FILE: {};
begin
   FILE := {the file};
   OPEN(FILE, IN_FILE, {the file_name});
   loop
      begin
         NEW_LINE; PUT("Enter DATA_RECORD Key: "); GET(DATA_RECORD_KEY);
         READ(FILE, DATA_RECORD, DATA_RECORD_KEY);
         exit;
      exception
         when END_ERROR => PUT("Invalid DATA_RECORD Key"); NEW_LINE;
      end;
   end loop;
   CLOSE(FILE);
   return DATA_RECORD_KEY;
exception
   when DEVICE_ERROR | END_ERROR | NAME_ERROR | STATUS_ERROR =>
      CLOSE(FILE); PUT("Data Base Inconsistent");
   when others => CLOSE(FILE); raise;
end QUERY_USER_FOR_KEY;
```

In addition to getting a key from the user, the cliche query_user_for_key checks the key to make sure that it is valid. In order to check the key, the cliche attempts to read the corresponding record from the file. If the read fails, then the user is asked for a different key.

Taking Care of Details

Both the code in Screen 35 and the cliche query_user_for_key specify that the file UNITS should be opened and closed. If nothing more were said this would cause the file UNITS to be opened twice and closed twice in Screen 36. In order to prevent this from happening, KBEmacs determines which files are required to be open and makes sure that each file is opened only once. Special provision is made for the opening and closing of files because this is an important efficiency issue in file oriented I/O.

s-F ***Fill the main_file_key with*** a query_user_for_key of UNITS. <end>

```
with CALENDAR, FUNCTIONS, MAINTENANCE_FILES, TEXT_IO;
use CALENDAR, FUNCTIONS, MAINTENANCE_FILES, TEXT_IO;
procedure UNIT_REPAIR_REPORT is
   use DEFECT_IO, REPAIR_IO, UNIT_IO, INT_IO;
   CURRENT_DATE: constant STRING := FORMAT_DATE(CLOCK);
   DEFECT: DEFECT_TYPE;
   REPAIR: REPAIR_TYPE;
   REPAIR_INDEX: REPAIR_INDEX_TYPE;
   REPORT: TEXT_IO.FILE_TYPE;
   TITLE: STRING(1..{});
   UNIT: UNIT_TYPE;
   UNIT_KEY: UNIT_KEY_TYPE;
   procedure CLEAN_UP is
      begin
         SET_OUTPUT(STANDARD_OUTPUT);
         CLOSE(DEFECTS); CLOSE(REPAIRS); CLOSE(UNITS); CLOSE(REPORT);
      exception
         when STATUS_ERROR => return;
      end CLEAN_UP;
begin
   OPEN(DEFECTS, IN_FILE, DEFECTS_NAME); OPEN(REPAIRS, IN_FILE, REPAIRS_NAME);
   OPEN(UNITS, IN_FILE, UNITS_NAME); CREATE(REPORT, OUT_FILE, "report.txt");
   TITLE := {the title};
   SET_OUTPUT(REPORT);
   NEW_LINE(4); SET_COL(20); PUT(CURRENT_DATE); NEW_LINE(2);
   SET_COL(13); PUT(TITLE); NEW_LINE(60);
   loop
      begin
         NEW_LINE; PUT("Enter UNIT Key: "); GET(UNIT_KEY);
         READ(UNITS, UNIT, UNIT_KEY);
         exit;
      exception
         when END_ERROR => PUT("Invalid UNIT Key"); NEW_LINE;
      end;
   end loop;
   READ(UNITS, UNIT, UNIT_KEY);
   REPAIR_INDEX := UNIT.REPAIR;
   while not NULL_INDEX(REPAIR_INDEX) loop
      READ(REPAIRS, REPAIR, REPAIR_INDEX);
      if LINE > 63 then
         NEW_PAGE; NEW_LINE; PUT("Page: "); PUT(INTEGER(PAGE-1), 3);
         SET_COL(13); PUT(TITLE); SET_COL(61); PUT(CURRENT_DATE); NEW_LINE(2);
         PUT("   Date     Defect   Description/Comment"); NEW_LINE(2);
      end if;
      READ(DEFECTS, DEFECT, REPAIR.DEFECT);
      PUT(FORMAT_DATE(REPAIR.DATE)); SET_COL(13); PUT(REPAIR.DEFECT);
      SET_COL(20); PUT(DEFECT.NAME); NEW_LINE;
      SET_COL(22); PUT(REPAIR.COMMENT); NEW_LINE;
      REPAIR_INDEX := REPAIR.NEXT;
   end loop;
   {the summary};
   CLEAN_UP;
exception
   when DEVICE_ERROR | END_ERROR | NAME_ERROR | STATUS_ERROR =>
      CLEAN_UP; PUT("Data Base Inconsistent");
```

KBEmacs (ADA) <KBE.DEMO>UNIT_REPAIR_REPORT.ADA Screen 36

The Knowledge-Based Command "Share"

The code in Screen 36 is inefficient in that it reads the same record in the file UNITS twice. The record is read twice because it is read for two different reasons by two different cliches. The cliche query_user_for_key reads the record in order to determine that the key is a valid key and the cliche chain_enumeration reads the record in order to obtain the index of the first repair record.

The existence of this problem illustrates that, although KBEmacs is capable of automatically sharing multiple opens and closes on a file, it is not capable of automatically sharing redundant code in general. However, KBEmacs provides a knowledge-based command "Share" which can assist a programmer in sharing code.

From the point of view of KBEmacs, the most difficult thing about sharing two computations is detecting that sharing might be useful. On the other hand it is quite easy for KBEmacs to check that two computations can be shared and to modify a program in order to effect a given sharing.

From the point of view of the programmer, the most difficult thing about sharing two computations is verifying that the two computations are in fact identical and can therefore actually be shared. The problem is that the existence of side-effects and the like can prevent the sharing of computations which otherwise appear identical.

The divergence in abilities between the programmer and KBEmacs with regard to sharing provides an opportunity for a useful division of labor. In Screen 37, the programmer determines that the two calls on READ of the file UNITS should be shared and asks KBEmacs to change the program so that they are shared. KBEmacs checks that the two calls on READ are in fact reading the same record and that they can be shared. It then modifies the program in order to achieve the sharing.

The sharing shown in Screen 37 is a relatively trivial example. KBEmacs is capable of handling much more complicated cases. There are two ways in which things can become more complex. First, if side-effects are involved, it can be much harder to determine whether the two computations to be shared can indeed by shared. KBEmacs is able to deal with side-effects because the system understands which functions have side-effects and the data flow in a plan takes side-effects into account. Second, sharing two computations can require more extensive changes in the program text. In particular, once one of the two computations has been removed, a variable usually has to be introduced so that the value created by the remaining computation can be used in two places. Here the changes are very simple because, in Screen 36, the procedure READ already forces the use of a variable and KBEmacs used the same variable for both READs. (KBEmacs tries to reuse variable names in logically related contexts in order to reduce the number of variables in a program.)

s-X Share the first READ and the second READ. <end>

```
with CALENDAR, FUNCTIONS, MAINTENANCE_FILES, TEXT_IO;
use CALENDAR, FUNCTIONS, MAINTENANCE_FILES, TEXT_IO;
procedure UNIT_REPAIR_REPORT is
   use DEFECT_IO, REPAIR_IO, UNIT_IO, INT_IO;
   CURRENT_DATE: constant STRING := FORMAT_DATE(CLOCK);
   DEFECT: DEFECT_TYPE;
   REPAIR: REPAIR_TYPE;
   REPAIR_INDEX: REPAIR_INDEX_TYPE;
   REPORT: TEXT_IO.FILE_TYPE;
   TITLE: STRING(1..{});
   UNIT: UNIT_TYPE;
   UNIT_KEY: UNIT_KEY_TYPE;
   procedure CLEAN_UP is
      begin
         SET_OUTPUT(STANDARD_OUTPUT);
         CLOSE(DEFECTS); CLOSE(REPAIRS); CLOSE(UNITS); CLOSE(REPORT);
      exception
         when STATUS_ERROR => return;
      end CLEAN_UP;
begin
   OPEN(DEFECTS, IN_FILE, DEFECTS_NAME); OPEN(REPAIRS, IN_FILE, REPAIRS_NAME);
   OPEN(UNITS, IN_FILE, UNITS_NAME); CREATE(REPORT, OUT_FILE, "report.txt");
   TITLE := {the title};
   SET_OUTPUT(REPORT);
   NEW_LINE(4); SET_COL(20); PUT(CURRENT_DATE); NEW_LINE(2);
   SET_COL(13); PUT(TITLE); NEW_LINE(60);
   loop
      begin
         NEW_LINE; PUT("Enter UNIT Key: "); GET(UNIT_KEY);
         READ(UNITS, UNIT, UNIT_KEY);
         exit;
      exception
         when END_ERROR => PUT("Invalid UNIT Key"); NEW_LINE;
      end;
   end loop;
   REPAIR_INDEX := UNIT.REPAIR;
   while not NULL_INDEX(REPAIR_INDEX) loop
      READ(REPAIRS, REPAIR, REPAIR_INDEX);
      if LINE > 63 then
         NEW_PAGE; NEW_LINE; PUT("Page: "); PUT(INTEGER(PAGE-1), 3);
         SET_COL(13); PUT(TITLE); SET_COL(61); PUT(CURRENT_DATE); NEW_LINE(2);
         PUT("   Date     Defect   Description/Comment"); NEW_LINE(2);
      end if;
      READ(DEFECTS, DEFECT, REPAIR.DEFECT);
      PUT(FORMAT_DATE(REPAIR.DATE)); SET_COL(13); PUT(REPAIR.DEFECT);
      SET_COL(20); PUT(DEFECT.NAME); NEW_LINE;
      SET_COL(22); PUT(REPAIR.COMMENT); NEW_LINE;
      REPAIR_INDEX := REPAIR.NEXT;
   end loop;
   {the summary};
   CLEAN_UP;
exception
   when DEVICE_ERROR | END_ERROR | NAME_ERROR | STATUS_ERROR =>
      CLEAN_UP; PUT("Data Base Inconsistent");
   when others => CLEAN_UP; raise;
```

```
KBEmacs (ADA) <KBE.DEMO>UNIT_REPAIR_REPORT.ADA
```

Screen 37

Specifying the Title

In Screen 38, the programmer uses the knowledge-based command "Fill" in order to specify the title to be used in the report. The title itself is expressed as a literal Ada expression in the command. A "Fill" command specifying a literal piece of program code is equivalent to using direct editing to textually replace the role with the program code. The expression is wrapped in parentheses so that the command language parser will recognize it as a single expression.

Using Common Sense

An interesting aspect of Screen 37 is that the title role happens to precede the GET of the UNIT_KEY. (These two computations are not ordered by control flow or data flow and KBEmacs chose to put the title first.) As a result, the "Fill" command in Screen 38 does not completely make sense. The problem is that it refers to the variable UNIT_KEY before it has been given a value.

KBEmacs tries to forestall problems in situations like this by not taking the exact order of program statements too literally. When the programmer specifies an expression to be inserted into a program, KBEmacs interprets each variable reference in the expression as a request to reference the associated value. If a variable does not have a value at the exact place where the insertion is to be made, then KBEmacs rearranges the program (within the limits of data flow and control flow constraints) in order to make an appropriate value available. Here, the GET of the UNIT_KEY is moved before the title.

Taking Care of Details

Another interesting change in Screen 38 is that once the computation of the title has been specified KBEmacs can compute the length of the string which stores the title. This is done based on the length of the literal string supplied by the programmer and on the fact that KBEmacs knows how long a UNIT_KEY is (by looking at the definition of the file UNITS in the package MAINTENANCE_FILES).

The fact that KBEmacs can determine the appropriate declaration for the variable TITLE shields the programmer from an irritating detail of the language Ada. Since there is no varying length string data type in standard Ada, the declaration of a local string variable (e.g., TITLE) must include the exact length of the string unless it is a constant (e.g., CURRENT_DATE).

s-X Fill the title with ("Report of Repairs on Unit " & UNIT_KEY). <end>

```
with CALENDAR, FUNCTIONS, MAINTENANCE_FILES, TEXT_IO;
use CALENDAR, FUNCTIONS, MAINTENANCE_FILES, TEXT_IO;
procedure UNIT_REPAIR_REPORT is
   use DEFECT_IO, REPAIR_IO, UNIT_IO, INT_IO;
   CURRENT_DATE: constant STRING := FORMAT_DATE(CLOCK);
   DEFECT: DEFECT_TYPE;
   REPAIR: REPAIR_TYPE;
   REPAIR_INDEX: REPAIR_INDEX_TYPE;
   REPORT: TEXT_IO.FILE_TYPE;
   TITLE: STRING(1..33);
   UNIT: UNIT_TYPE;
   UNIT_KEY: UNIT_KEY_TYPE;
   procedure CLEAN_UP is
      begin
         SET_OUTPUT(STANDARD_OUTPUT);
         CLOSE(DEFECTS); CLOSE(REPAIRS); CLOSE(UNITS); CLOSE(REPORT);
      exception
         when STATUS_ERROR => return;
      end CLEAN_UP;
begin
   OPEN(DEFECTS, IN_FILE, DEFECTS_NAME); OPEN(REPAIRS, IN_FILE, REPAIRS_NAME);
   OPEN(UNITS, IN_FILE, UNITS_NAME); CREATE(REPORT, OUT_FILE, "report.txt");
   loop
      begin
         NEW_LINE; PUT("Enter UNIT Key: "); GET(UNIT_KEY);
         READ(UNITS, UNIT, UNIT_KEY);
         exit;
      exception
         when END_ERROR => PUT("Invalid UNIT Key"); NEW_LINE;
      end;
   end loop;
   TITLE := "Report of Repairs on Unit " & UNIT_KEY;
   SET_OUTPUT(REPORT);
   NEW_LINE(4); SET_COL(20); PUT(CURRENT_DATE); NEW_LINE(2);
   SET_COL(13); PUT(TITLE); NEW_LINE(60);
   REPAIR_INDEX := UNIT.REPAIR;
   while not NULL_INDEX(REPAIR_INDEX) loop
      READ(REPAIRS, REPAIR, REPAIR_INDEX);
      if LINE > 63 then
         NEW_PAGE; NEW_LINE; PUT("Page: "); PUT(INTEGER(PAGE-1), 3);
         SET_COL(13); PUT(TITLE); SET_COL(61); PUT(CURRENT_DATE); NEW_LINE(2);
         PUT("   Date     Defect   Description/Comment"); NEW_LINE(2);
      end if;
      READ(DEFECTS, DEFECT, REPAIR.DEFECT);
      PUT(FORMAT_DATE(REPAIR.DATE)); SET_COL(13); PUT(REPAIR.DEFECT);
      SET_COL(20); PUT(DEFECT.NAME); NEW_LINE;
      SET_COL(22); PUT(REPAIR.COMMENT); NEW_LINE;
      REPAIR_INDEX := REPAIR.NEXT;
   end loop;
   {the summary};
   CLEAN_UP;
exception
   when DEVICE_ERROR | END_ERROR | NAME_ERROR | STATUS_ERROR =>
      CLEAN_UP; PUT("Data Base Inconsistent");
   when others => CLEAN_UP; raise;
```

Finishing the Program

In Screen 39, the programmer completes the program UNIT_REPAIR_REPORT by removing the summary role. He does this by using the abbreviated command "s-R" which offers to remove the first unfilled role following the current position of the cursor. The line_limit is changed from 63 to 64 since zero lines are now required for the summary whereas the default assumption used by the constraint function SIZE_IN_LINES in the screens above was that the summary would produce one line of output. The programmer then tells KBEmacs that the program is complete and compiles the program so that it can be tested.

Evaluating the Commands Used

Consider the set of knowledge-based commands (shown below) which were used in order to implement the program UNIT_REPAIR_REPORT.

```
Define a simple_report procedure UNIT_REPAIR_REPORT.
Fill the enumerator with a chain_enumeration of UNITS and REPAIRS.
Fill the main_file_key with a query_user_for_key of UNITS.
Share the first READ and the second READ.
Fill the title with ("Report of Repairs on Unit " & UNIT_KEY).
Remove the summary.
```

Comparing these commands with the hypothetical directions for a human assistant shows that they are virtually identical. The only real difference is that KBEmacs was not smart enough to automatically share the two calls on READ.

```
Define a simple report program UNIT_REPAIR_REPORT.  Enumerate the chain of
repairs associated with a unit record, printing each one.  Query the user for
the key (UNIT_KEY) of the unit record to start from.  Print the title
("Report of Repairs on Unit " & UNIT_KEY).  Do not print a summary.
```

It is also interesting to compare the KBEmacs commands used to construct the program UNIT_REPAIR_REPORT with the directions (reproduced below) which were used to create the program REPORT-TIMINGS in the last chapter.

```
Define a program REPORT-TIMINGS with a parameter TIMINGS.
Insert a simple-report.
Fill the title with "Report of Reaction Timings (in msec.)".
Fill the enumerator with a list-enumeration of TIMINGS.
Remove the column-headings.
Fill the print-item with a print-out of "~&~5X~8:D".
Fill the summary with (FORMAT REPORT "~2&~{mean:~8:D   (deviation: ~:D)~}"
                                     (MEAN-AND-DEVIATION TIMINGS)).
```

This comparison leads to two points. First, the commands for UNIT_REPAIR_REPORT are significantly simpler due to the action of the constraints defined as part of the cliche simple_report and the existence of the cliches print_repair and print_repair_headings. Second, the language independence of KBEmacs is illustrated by the fact that the two sets of commands are really very similar. If it were not for the inclusion of a few literal pieces of program code, there would be no way to know that the programs produced by the two sets of commands were coded in two different programming languages.

s-R ***Remove the summary.*** <end>
s-<end> ***Finish editing the procedure UNIT_REPAIR_REPORT. <end>***
c-shift-C ***Compiling UNIT_REPAIR_REPORT***

```
with CALENDAR, FUNCTIONS, MAINTENANCE_FILES, TEXT_IO;
use CALENDAR, FUNCTIONS, MAINTENANCE_FILES, TEXT_IO;
procedure UNIT_REPAIR_REPORT is
   use DEFECT_IO, REPAIR_IO, UNIT_IO, INT_IO;
   CURRENT_DATE: constant STRING := FORMAT_DATE(CLOCK);
   DEFECT: DEFECT_TYPE;
   REPAIR: REPAIR_TYPE;
   REPAIR_INDEX: REPAIR_INDEX_TYPE;
   REPORT: TEXT_IO.FILE_TYPE;
   TITLE: STRING(1..33);
   UNIT: UNIT_TYPE;
   UNIT_KEY: UNIT_KEY_TYPE;
   procedure CLEAN_UP is
      begin
         SET_OUTPUT(STANDARD_OUTPUT);
         CLOSE(DEFECTS); CLOSE(REPAIRS); CLOSE(UNITS); CLOSE(REPORT);
      exception
         when STATUS_ERROR => return;
      end CLEAN_UP;
begin
   OPEN(DEFECTS, IN_FILE, DEFECTS_NAME); OPEN(REPAIRS, IN_FILE, REPAIRS_NAME);
   OPEN(UNITS, IN_FILE, UNITS_NAME); CREATE(REPORT, OUT_FILE, "report.txt");
   loop
      begin
         NEW_LINE; PUT("Enter UNIT Key: "); GET(UNIT_KEY);
         READ(UNITS, UNIT, UNIT_KEY);
         exit;
      exception
         when END_ERROR => PUT("Invalid UNIT Key"); NEW_LINE;
      end;
   end loop;
   TITLE := "Report of Repairs on Unit " & UNIT_KEY;
   SET_OUTPUT(REPORT);
   NEW_LINE(4); SET_COL(20); PUT(CURRENT_DATE); NEW_LINE(2);
   SET_COL(13); PUT(TITLE); NEW_LINE(60);
   REPAIR_INDEX := UNIT.REPAIR;
   while not NULL_INDEX(REPAIR_INDEX) loop
      READ(REPAIRS, REPAIR, REPAIR_INDEX);
      if LINE > 64 then
         NEW_PAGE; NEW_LINE; PUT("Page: "); PUT(INTEGER(PAGE-1), 3);
         SET_COL(13); PUT(TITLE); SET_COL(61); PUT(CURRENT_DATE); NEW_LINE(2);
         PUT("   Date      Defect    Description/Comment"); NEW_LINE(2);
      end if;
      READ(DEFECTS, DEFECT, REPAIR.DEFECT);
      PUT(FORMAT_DATE(REPAIR.DATE)); SET_COL(13); PUT(REPAIR.DEFECT);
      SET_COL(20); PUT(DEFECT.NAME); NEW_LINE;
      SET_COL(22); PUT(REPAIR.COMMENT); NEW_LINE;
      REPAIR_INDEX := REPAIR.NEXT;
   end loop;
   CLEAN_UP;
exception
   when DEVICE_ERROR | END_ERROR | NAME_ERROR | STATUS_ERROR =>
      CLEAN_UP; PUT("Data Base Inconsistent");
   when others => CLEAN_UP; raise;
end UNIT_REPAIR_REPORT;
```

KBEmacs (ADA) <KBE.DEMO>UNIT_REPAIR_REPORT.ADA

Screen 39

Documentation

In Screen 40, the programmer asks KBEmacs to create a comment for the program. This comment is created in exactly the same way as the comment for a Lisp program. The only difference is that the Ada comment delimiter is used, and literal pieces of code in the comment are rendered in Ada.

s-; *Comment the procedure UNIT_REPAIR_REPORT. <end>*

```
-- The procedure UNIT_REPAIR_REPORT is a simple_report:
--   The file_name is "report.txt".
--   The title is ("Report of Repairs on Unit " & UNIT_KEY).
--   The enumerator is a chain_enumeration.
--      It enumerates the chain records in REPAIRS starting from the
--      the header record indexed by UNIT_KEY.
--   The column_headings are a print_repair_headings.
--      It prints headings for printing repair records.
--   The print_item is a print_repair.
--      It prints out the fields of REPAIR.
--   There is no summary.
□
with CALENDAR, FUNCTIONS, MAINTENANCE_FILES, TEXT_IO;
use CALENDAR, FUNCTIONS, MAINTENANCE_FILES, TEXT_IO;
procedure UNIT_REPAIR_REPORT is
   use DEFECT_IO, REPAIR_IO, UNIT_IO, INT_IO;
   CURRENT_DATE: constant STRING := FORMAT_DATE(CLOCK);
   DEFECT: DEFECT_TYPE;
   REPAIR: REPAIR_TYPE;
   REPAIR_INDEX: REPAIR_INDEX_TYPE;
   REPORT: TEXT_IO.FILE_TYPE;
   TITLE: STRING(1..33);
   UNIT: UNIT_TYPE;
   UNIT_KEY: UNIT_KEY_TYPE;
   procedure CLEAN_UP is
      begin
         SET_OUTPUT(STANDARD_OUTPUT);
         CLOSE(DEFECTS); CLOSE(REPAIRS); CLOSE(UNITS); CLOSE(REPORT);
      exception
         when STATUS_ERROR => return;
      end CLEAN_UP;
begin
   OPEN(DEFECTS, IN_FILE, DEFECTS_NAME); OPEN(REPAIRS, IN_FILE, REPAIRS_NAME);
   OPEN(UNITS, IN_FILE, UNITS_NAME); CREATE(REPORT, OUT_FILE, "report.txt");
   loop
      begin
         NEW_LINE; PUT("Enter UNIT Key: "); GET(UNIT_KEY);
         READ(UNITS, UNIT, UNIT_KEY);
         exit;
      exception
         when END_ERROR => PUT("Invalid UNIT Key"); NEW_LINE;
      end;
   end loop;
   TITLE := "Report of Repairs on Unit " & UNIT_KEY;
   SET_OUTPUT(REPORT);
   NEW_LINE(4); SET_COL(20); PUT(CURRENT_DATE); NEW_LINE(2);
   SET_COL(13); PUT(TITLE); NEW_LINE(60);
   REPAIR_INDEX := UNIT.REPAIR;
   while not NULL_INDEX(REPAIR_INDEX) loop
      READ(REPAIRS, REPAIR, REPAIR_INDEX);
      if LINE > 64 then
         NEW_PAGE; NEW_LINE; PUT("Page: "); PUT(INTEGER(PAGE-1), 3);
         SET_COL(13); PUT(TITLE); SET_COL(61); PUT(CURRENT_DATE); NEW_LINE(2);
         PUT("   Date     Defect   Description/Comment"); NEW_LINE(2);
      end if;
```

Testing

As an adjunct to KBEmacs, a minimal interpreter for a subset of Ada has been implemented so that Ada programs can be tested. In Screen 41, the programmer tests the program UNIT_REPAIR_REPORT. The programmer first uses the procedure CREATE_TEST_RECORDS (see Appendix B) to create a test data base containing the records shown in Figure 7. He then runs the program UNIT_REPAIR_REPORT. In addition to creating a report he tests what happens when the user inputs an invalid key to the file UNITS. Finally, the programmer uses the program DISPLAY_REPORT (also shown in Appendix B) to display the report produced.

<select> A

```
CREATE_TEST_RECORDS;
UNIT_REPAIR_REPORT;
Enter Unit Key: GA2-341
Invalid Unit Key

Enter Unit Key: GA2-342

DISPLAY_REPORT;
< 4 blank lines>
                    5/27/1985

            Report of Repairs on Unit GA2-342
<59 blank lines>
<page>
Page:   1   Report of Repairs on Unit GA2-342                5/27/1985

   Date      Defect   Description/Comment

 3/19/1985   GA-32  Clogged gas injection port
                      Port Diameter seems below specs.
 1/23/1985   GA-11  Control board cold solder joint
                      Took two days to find.
 9/14/1983   GA-32  Clogged gas injection port
                      Probably caused by humidity.
```

A Large Program

In the last part of the scenario, the programmer constructs a complex report program named MODEL_DEFECTS_REPORT. This program produces a summary report of the defects which have occurred on units of a given model since a given date. The following directions might be used to tell a person how to write the program.

```
Define a simple report program MODEL_DEFECTS_REPORT.  Query the user for a
key of a record in the file MODELS and for a starting date.  Enumerate all of
the repairs performed on all of the units and select the repairs which have
been performed on units of the specified model since the start date.  Print a
summarized report showing how many times each kind of defect occurs in these
repairs.  Print a title showing the specified model and the starting date.
Do not print a summary.
```

There are six key concepts which underlie the directions above: how to print a simple report, how to query the user for a key, how to query the user for a date, how to enumerate all of the repairs, how to select a subset of them, and how to summarize them showing the frequency of each defect. They first two of these concepts were used in the program UNIT_REPAIR_REPORT. The remaining four will be discussed in detail below.

The Cliche File_Selection

The cliche file_selection (shown on the next page) creates a file of selected keys to records in another file. The cliche is a combination of three more basic cliches: enumeration, selection, and file_accumulation (all shown in Appendix A). The cliche enumerates some series of values, selects some subseries of these values, and writes a file of keys corresponding to these values.

The cliche has four principal roles. The *source_file* is the file to which keys are being selected. The *enumerator* enumerates some series of records. It is expected that these records will either be the records in the source_file, or other records which contain keys to the source_file. The *selection_test* selects the enumerated records which are of interest. The *record_key_accessor* determines what source_file key corresponds to each selected record.

The cliche includes a declaration for the file SELECTIONS which is used to contain the selected keys. The I/O functions for this file are an instantiation of the standard Ada generic package DIRECT_IO. An output role is used to specify that the file SELECTIONS is the logical output of the cliche.

The constraint in the cliche is used to rename the role record_key_accessor so that it will have a more mnemonic name when the cliche is instantiated. For example, if the source_file is specified to be the file DEFECTS then the record_key_accessor role will be renamed the defect_key_accessor.

```
with DIRECT_IO;
cliche FILE_SELECTION is
   primary roles SOURCE_FILE;
   described roles SOURCE_FILE, SELECTION_TEST, RECORD_KEY_ACCESSOR;
   comment "creates a file containing selected record keys for
            {the source_file}";
   constraints
      RENAME("RECORD_KEY", RECORD_KEY_ROOT({the source_file}));
   end constraints;

   package SELECTION_IO is new DIRECT_IO(RECORD_KEY_TYPE);
   use SELECTION_IO;
   SELECTIONS: SELECTION_IO.FILE_TYPE;

   DATA: {};
   DATUM: {};
begin
   DATA := {the source_file};
   CREATE(SELECTIONS, INOUT_FILE);
   while not {the empty_test of the enumerator}(DATA) loop
      DATUM := {the element_accessor of the enumerator}(DATA);
      if {the selection_test}(DATUM) then
         WRITE(SELECTIONS, {the record_key_accessor}(DATUM));
      end if;
      DATA := {the step of the enumerator}(DATA);
   end loop;
   {SELECTIONS, the output selection_file};
   CLOSE(SELECTIONS);
exception
   when DEVICE_ERROR | END_ERROR | NAME_ERROR | STATUS_ERROR =>
      CLOSE(SELECTIONS); PUT("Data Base Inconsistent");
   when others => CLOSE(SELECTIONS); raise;
end FILE_SELECTION;
```

The Cliche File_Summarization

```
with DIRECT_IO;
cliche FILE_SUMMARIZATION is
   primary roles SOURCE_FILE;
   described roles SOURCE_FILE;
   comment "creates a file summarizing the frequency of occurrence of
            the record keys stored in {the source file}";
   constraints
      RENAME("SOURCE_RECORD", SINGULAR_FORM({the source_file}));
      RENAME("RECORD_KEY", RECORD_KEY_ROOT({the source_file}));
      DEFAULT({the source_file_name},
              CORRESPONDING_FILE_NAME({the source_file}));
   end constraints;

   type SUMMARY_TYPE is
      record
         COUNT: INTEGER;
         KEY: RECORD_KEY_TYPE;
      end record;
   package SUMMARY_IO is new DIRECT_IO(SUMMARY_TYPE);
   use SUMMARY_IO;
   SUMMARIES: SUMMARY_IO.FILE_TYPE;

   SOURCE_FILE: {};

   function BUILD_SUMMARY(COUNT: INTEGER; KEY: RECORD_KEY_TYPE)
                              return SUMMARY_TYPE is
      begin return SUMMARY_TYPE'(COUNT, KEY); end BUILD_SUMMARY;
   function SUMMARY_GREATER_THAN(X: SUMMARY_TYPE; Y: SUMMARY_TYPE)
                                     return BOOLEAN is
      begin return X.COUNT > Y.COUNT; end SUMMARY_GREATER_THAN;
   procedure SORT_SOURCE_RECORDS is
      new SORT_FILE(RECORD_KEY_TYPE, SOURCE_RECORD_IO.FILE_TYPE,
                    SOURCE_RECORD_IO.POSITIVE_COUNT);
   procedure SORT_SUMMARIES is
      new SORT_FILE(SUMMARY_TYPE, SUMMARY_IO.FILE_TYPE,
                    SUMMARY_IO.POSITIVE_COUNT, SUMMARY_GREATER_THAN);
   procedure SUMMARIZE_SOURCE_RECORDS is
      new SUMMARIZE_FILE(RECORD_KEY_TYPE, SOURCE_RECORD_IO.FILE_TYPE,
                         SUMMARY_TYPE, SUMMARY_IO.FILE_TYPE, BUILD_SUMMARY);
   procedure CLEAN_UP is
      begin
         CLOSE(SOURCE_FILE); CLOSE(SUMMARIES);
      exception
         when STATUS_ERROR => return;
      end CLEAN_UP;
begin
   SOURCE_FILE := {the source_file};
   OPEN(SOURCE_FILE, INOUT_FILE, {the source_file_name});
   CREATE(SUMMARIES, INOUT_FILE);
   SORT_SOURCE_RECORDS(SOURCE_FILE);
   SUMMARIZE_SOURCE_RECORDS(SOURCE_FILE, SUMMARIES);
   SORT_SUMMARIES(SUMMARIES);
   {SUMMARIES, the output summaries};
   CLEAN_UP;
exception
   when DEVICE_ERROR | END_ERROR | NAME_ERROR | STATUS_ERROR =>
      CLEAN_UP; PUT("Data Base Inconsistent");
   when others => CLEAN_UP; raise;
end FILE_SUMMARIZATION;
```

The cliche file_summarization (shown on the previous page) takes in a source file of values and creates a summary file which shows how many times each value occurs in the source file. Each record in the summary file contains one of the values from the source file and a count of how many times that value occurs in the source file. The records in the summary file are sorted with the most frequent values first. As an example of what the cliche does, suppose that the source file contained the values "`A A B A C A C`". The summary file would then contain the records "`(4 A) (2 C) (1 B)`".

The cliche has only one principal role: the *source_file*. The summary file is the logical output of the cliche. The cliche works by sorting the source file so that identical items become adjacent, creating the summary file by counting how many times each value exists in the source file, and then sorting the summary file.

The cliche depends heavily on the generic procedures `SORT_FILE` and `SUMMARIZE_FILE` which are defined in the package `FUNCTIONS` (see Appendix B). Most of the code for the cliche consists of declarations which are required in order to instantiate these procedures. The body of the code is a series of calls on these instantiations.

The cliche file_summarization is another example of the fact that KBEmacs uses existing mechanisms whenever possible. Since generic procedures are fully adequate to represent the actions of sorting a file and performing the actual summarization, there is not reason to open code these operations in the cliche. This simplifies both the cliche and the programs which use the cliche. The aspects of file summarization which cannot be represented as a generic (i.e., the declaration of the summary file records and the functions which operate on them) are retained in the cliche.

Constructing the Program

In Screen 42, the programmer begins the construction of the program `MODEL_DEFECTS_REPORT` by creating a file to hold the program, and then creating an empty definition for the program. He then specifies the basic structure of the program by combining the four cliches simple_report, file_enumeration, file_selection, and file_summarization.

The code which results from combining these cliches is too large to show in one screen. The declarations in the program are shown in Screen 42a. Although some of the declarations come from the cliches file_selection and simple_report, most of the declarations come from the cliche file_summarization.

The body of the program is shown in Screen 42b. The body is divided into three logical parts. The first loop comes from the cliche file_selection. The next three lines come from the cliche file_summarization. The rest of the body is a simple_report whose enumerator is a file_enumeration. The print_item and column_headings have been automatically filled in by the functions `CORRESPONDING_PRINTING` and `CORRESPONDING_HEADINGS`. Since no cliche is defined for the printing of a summary, these functions generated the appropriate code based on the declaration of the file `SUMMARIES`.

```
c-X c-F Find file <KBE.DEMO>MODEL_DEFECTS_REPORT.ADA
(New File)
s-X Define a procedure MODEL_DEFECTS_REPORT. <end>
s-X Insert a simple_report of a file_enumeration of a file_summarization of
      a file_selection of DEFECTS. <end>
```

```
with CALENDAR, FUNCTIONS, DIRECT_IO, MAINTENANCE_FILES, TEXT_IO;
use CALENDAR, FUNCTIONS, MAINTENANCE_FILES, TEXT_IO;
procedure MODEL_DEFECTS_REPORT is
   use DEFECT_IO, INT_IO;

   package SELECTION_IO is new DIRECT_IO(DEFECT_KEY_TYPE);
   use SELECTION_IO;
   SELECTIONS: SELECTION_IO.FILE_TYPE;

   type SUMMARY_TYPE is
      record
         COUNT: INTEGER;
         KEY: DEFECT_KEY_TYPE;
      end record;
   package SUMMARY_IO is new DIRECT_IO(SUMMARY_TYPE);
   use SUMMARY_IO;
   SUMMARIES: SUMMARY_IO.FILE_TYPE;

   CURRENT_DATE: constant STRING := FORMAT_DATE(CLOCK);
   DATA: {};
   DATUM: {};
   DEFECT: DEFECT_TYPE;
   REPORT: TEXT_IO.FILE_TYPE;
   SUMMARY: SUMMARY_TYPE;
   TITLE: STRING(1..{});

   function BUILD_SUMMARY(COUNT: INTEGER; KEY: DEFECT_KEY_TYPE)
                             return SUMMARY_TYPE is
      begin return SUMMARY_TYPE'(COUNT, KEY); end BUILD_SUMMARY;
   function SUMMARY_GREATER_THAN(X: SUMMARY_TYPE; Y: SUMMARY_TYPE)
                                    return BOOLEAN is
      begin return X.COUNT > Y.COUNT; end SUMMARY_GREATER_THAN;
   procedure SORT_SELECTIONS is
      new SORT_FILE(DEFECT_KEY_TYPE, SELECTION_IO.FILE_TYPE,
                    SELECTION_IO.POSITIVE_COUNT);
   procedure SORT_SUMMARIES is
      new SORT_FILE(SUMMARY_TYPE, SUMMARY_IO.FILE_TYPE,
                    SUMMARY_IO.POSITIVE_COUNT,SUMMARY_GREATER_THAN);
   procedure SUMMARIZE_SELECTIONS is
      new SUMMARIZE_FILE(DEFECT_KEY_TYPE, SELECTION_IO.FILE_TYPE, SUMMARY_TYPE,
                         SUMMARY_IO.FILE_TYPE, BUILD_SUMMARY);
   procedure CLEAN_UP is
      begin
         SET_OUTPUT(STANDARD_OUTPUT);
         CLOSE(DEFECTS); CLOSE(REPORT); CLOSE(SELECTIONS); CLOSE(SUMMARIES);
      exception
         when STATUS_ERROR => return;
      end CLEAN_UP;
```

```
KBEmacs (ADA) <KBE.DEMO>MODEL_DEFECTS_REPORT.ADA
```

Screen 42a

```
begin
   OPEN(DEFECTS, IN_FILE, DEFECTS_NAME); CREATE(REPORT, OUT_FILE, "report.txt");
   CREATE(SELECTIONS, INOUT_FILE); CREATE(SUMMARIES, INOUT_FILE);
   DATA := DEFECTS;
   while not {the empty_test of the enumerator}(DATA) loop
      DATUM := {the element_accessor of the enumerator}(DATA);
      if {the selection_test}(DATUM) then
         WRITE(SELECTIONS, {the defect_key_accessor}(DATUM));
      end if;
      DATA := {the step of the enumerator}(DATA);
   end loop;
   SORT_SELECTIONS(SELECTIONS);
   SUMMARIZE_SELECTIONS(SELECTIONS, SUMMARIES);
   SORT_SUMMARIES(SUMMARIES);
   SET_OUTPUT(REPORT);
   TITLE := {the title};
   NEW_LINE(4); SET_COL(20); PUT(CURRENT_DATE); NEW_LINE(2);
   SET_COL(13); PUT(TITLE); NEW_LINE(60);
   while not END_OF_FILE(SUMMARIES) loop
      READ(SUMMARIES, SUMMARY);
      if LINE > 54 then
         NEW_PAGE; NEW_LINE; PUT("Page: "); PUT(INTEGER(PAGE-1), 3);
         SET_COL(13); PUT(TITLE); SET_COL(61); PUT(CURRENT_DATE); NEW_LINE(2);
         PUT("  # Key    Name"); NEW_LINE(2);
      end if;
      READ(DEFECTS, DEFECT, SUMMARY.KEY);
      PUT(SUMMARY.COUNT, 3); SET_COL(5); PUT(SUMMARY.KEY);
      SET_COL(12); PUT(DEFECT.NAME); NEW_LINE;
   end loop;
   {the summary};
   CLEAN_UP;
exception
   when DEVICE_ERROR | END_ERROR | NAME_ERROR | STATUS_ERROR =>
      CLEAN_UP; PUT("Data Base Inconsistent");
   when others => CLEAN_UP; raise;
end MODEL_DEFECTS_REPORT;
```

KBEmacs (ADA) <KBE.DEMO>MODEL_DEFECTS_REPORT.ADA

Screen 42b

The Cliche All_Chains_Enumeration

In Screen 43, the programmer fills in the enumerator of the file_selection. He does this using the cliche all_chains_enumeration (see Appendix A). This cliche has two primary roles: the *main_file* and the *chain_file*. It enumerates all of the records in the chain_file which are in chains that start from any of the records in the main_file. It does this by enumerating all of the records in the main_file and enumerating the chain of records starting with each of the main_file records. In as much as this is the case, the cliche is a combination of the cliches file_enumeration and chain_enumeration.

The highlighting in Screen 43 indicates the code which comes from the cliche all_chains_enumeration. (In the interest of brevity, the declarations in the program `MODEL_DEFECTS_REPORT` will not be shown in the next few screens. They will be discussed again at the end of the scenario.)

The primary area of complexity in the cliche all_chains_enumeration is that it is, in a sense, turned inside-out. One might expect that the most natural way to write the cliche would be as a nested loop with the outer loop enumerating main_file records and the inner loop enumerating chain_file records. However, the cliche is actually written so that the inner loop enumerates main_file records and the outer loop enumerates chain_file records. It is important that it be done this way so that the cliche will fit into all of the places where a simple linear enumerator is expected.

```
s-F Fill the enumerator with an all_chains_enumeration of UNITS and REPAIRS. <end>
```

```
begin
   OPEN(DEFECTS, IN_FILE, DEFECTS_NAME); OPEN(REPAIRS, IN_FILE, REPAIRS_NAME);
   OPEN(UNITS, IN_FILE, UNITS_NAME); CREATE(REPORT, OUT_FILE, "report.txt");
   CREATE(SELECTIONS, INOUT_FILE); CREATE(SUMMARIES, INOUT_FILE);
   REPAIR_INDEX := NULL_INDEX;
   loop
      while NULL_INDEX(REPAIR_INDEX) and not END_OF_FILE(UNITS) loop
         READ(UNITS, UNIT);
         REPAIR_INDEX := UNIT.REPAIR;
      end loop;
      exit when NULL_INDEX(REPAIR_INDEX);
      READ(REPAIRS, REPAIR, REPAIR_INDEX);
      if {the selection_test}(REPAIR) then
         WRITE(SELECTIONS, {the defect_key_accessor}(REPAIR));
      end if;
      REPAIR_INDEX := REPAIR.NEXT;
   end loop;
   SORT_SELECTIONS(SELECTIONS);
   SUMMARIZE_SELECTIONS(SELECTIONS, SUMMARIES);
   SORT_SUMMARIES(SUMMARIES);
   SET_OUTPUT(REPORT);
   TITLE := {the title};
   NEW_LINE(4); SET_COL(20); PUT(CURRENT_DATE); NEW_LINE(2);
   SET_COL(13); PUT(TITLE); NEW_LINE(60);
   while not END_OF_FILE(SUMMARIES) loop
      READ(SUMMARIES, SUMMARY);
      if LINE > 54 then
         NEW_PAGE; NEW_LINE; PUT("Page: "); PUT(INTEGER(PAGE-1), 3);
         SET_COL(13); PUT(TITLE); SET_COL(61); PUT(CURRENT_DATE); NEW_LINE(2);
         PUT("  # Key    Name"); NEW_LINE(2);
      end if;
      READ(DEFECTS, DEFECT, SUMMARY.KEY);
      PUT(SUMMARY.COUNT, 3); SET_COL(5); PUT(SUMMARY.KEY);
      SET_COL(12); PUT(DEFECT.NAME); NEW_LINE;
   end loop;
   {the summary};
   CLEAN_UP;
exception
   when DEVICE_ERROR | END_ERROR | NAME_ERROR | STATUS_ERROR =>
      CLEAN_UP; PUT("Data Base Inconsistent");
   when others => CLEAN_UP; raise;
end MODEL_DEFECTS_REPORT;
```

```
KBEmacs (ADA) <KBE.DEMO>MODEL_DEFECTS_REPORT.ADA
```

Screen 43

Specifying the Selection_Test

In Screen 44, the programmer uses direct editing to fill in the selection_test. In order to specify the selection_test, he makes use of several inline cliche references. The cliche query_user_for_key has been discussed above.

The function QUERY_USER_FOR_DATE (defined in the package FUNCTIONS) takes a prompt string as its argument and reads in a value of type TIME. This computation is implemented as a function rather than a cliche because it can be easily and efficiently represented that way. As always, cliches are only used when other mechanisms are inadequate.

The cliche pre_loop_computation (shown below) is needed in order to specify that the user is to be queried only once before the loop starts instead of on each cycle of the loop. The cliche pre_loop_computation is has one principal role: the *item*. It makes this value available as an output on each cycle of a loop. The cliche is convenient because it allows the embedded cliche references in Screen 44 to be written as simple expressions. Without this cliche, the programmer would have to insert the code for querying the user before the beginning of the loop and explicitly introduce variables to carry the values into the loop.

```
cliche PRE_LOOP_COMPUTATION is
   primary roles ITEM;
   described roles ITEM;
   comment "makes the value {the item} available inside of a loop";
   constraints
      RENAME("DATA_VALUE", SUGGEST_VARIABLE_NAME({the item}));
   end constraints;

   DATA_VALUE: {};
begin
   DATA_VALUE := {the item};
   loop
      {DATA_VALUE, the output data_value};
   end loop;
end PRE_LOOP_COMPUTATION;
```

In Screen 44, the programmer also specifies that the defect_key_accessor should access the DEFECT field of the selected repair records.

s-N *direct editing*

```
begin
   OPEN(DEFECTS, IN_FILE, DEFECTS_NAME); OPEN(REPAIRS, IN_FILE, REPAIRS_NAME);
   OPEN(UNITS, IN_FILE, UNITS_NAME); CREATE(REPORT, OUT_FILE, "report.txt");
   CREATE(SELECTIONS, INOUT_FILE); CREATE(SUMMARIES, INOUT_FILE);
   REPAIR_INDEX := NULL_INDEX;
   loop
      while NULL_INDEX(REPAIR_INDEX) and not END_OF_FILE(UNITS) loop
         READ(UNITS, UNIT);
         REPAIR_INDEX := UNIT.REPAIR;
      end loop;
      exit when NULL_INDEX(REPAIR_INDEX);
      READ(REPAIRS, REPAIR, REPAIR_INDEX);
      if UNIT.MODEL = {a pre_loop_computation of a query_user_for_key of MODELS}
         and REPAIR.DATE >=
             {a pre_loop_computation of
                QUERY_USER_FOR_DATE("Start of Reporting Period")} then
         WRITE(SELECTIONS, REPAIR.DEFECT);
      end if;
      REPAIR_INDEX := REPAIR.NEXT;
   end loop;
   SORT_SELECTIONS(SELECTIONS);
   SUMMARIZE_SELECTIONS(SELECTIONS, SUMMARIES);
   SORT_SUMMARIES(SUMMARIES);
   SET_OUTPUT(REPORT);
   TITLE := {the title};
   NEW_LINE(4); SET_COL(20); PUT(CURRENT_DATE); NEW_LINE(2);
   SET_COL(13); PUT(TITLE); NEW_LINE(60);
   while not END_OF_FILE(SUMMARIES) loop
      READ(SUMMARIES, SUMMARY);
      if LINE > 54 then
         NEW_PAGE; NEW_LINE; PUT("Page: "); PUT(INTEGER(PAGE-1), 3);
         SET_COL(13); PUT(TITLE); SET_COL(61); PUT(CURRENT_DATE); NEW_LINE(2);
         PUT("  # Key    Name"); NEW_LINE(2);
      end if;
      READ(DEFECTS, DEFECT, SUMMARY.KEY);
      PUT(SUMMARY.COUNT, 3); SET_CDL(5); PUT(SUMMARY.KEY);
      SET_COL(12); PUT(DEFECT.NAME); NEW_LINE;
   end loop;
   {the summary};
   CLEAN_UP;
exception
   when DEVICE_ERROR | END_ERROR | NAME_ERROR | STATUS_ERROR =>
      CLEAN_UP; PUT("Data Base Inconsistent");
   when others => CLEAN_UP; raise;
end MODEL_DEFECTS_REPORT;
```

Analyzing Direct Editing Done by the Programmer

In Screen 45, the programmer uses the knowledge-based command "Analyze" in order to see the ramifications of the direct editing he performed in Screen 44. KBEmacs inserts code for querying the user and two variables which carry the values into the loop.

```
s-A Analyze the procedure MODEL_DEFECTS_REPORT. <end>
```

```
begin
   OPEN(DEFECTS, IN_FILE, DEFECTS_NAME); OPEN(MODELS, IN_FILE, MODELS_NAME);
   OPEN(REPAIRS, IN_FILE, REPAIRS_NAME); OPEN(UNITS, IN_FILE, UNITS_NAME);
   CREATE(REPORT, OUT_FILE, "report.txt"); CREATE(SELECTIONS, INOUT_FILE);
   CREATE(SUMMARIES, INOUT_FILE);
   loop
      begin
         NEW_LINE; PUT("Enter Model Key: ");
         GET(MODEL_KEY);
         READ(MODELS, MODEL, MODEL_KEY);
         exit;
      exception
         when END_ERROR => PUT("Invalid MODEL Key"); NEW_LINE;
      end;
   end loop;
   DATE := QUERY_USER_FOR_DATE("Start of Reporting Period");
   REPAIR_INDEX := NULL_INDEX;
   loop
      while NULL_INDEX(REPAIR_INDEX) and not END_OF_FILE(UNITS) loop
         READ(UNITS, UNIT);
         REPAIR_INDEX := UNIT.REPAIR;
      end loop;
      exit when NULL_INDEX(REPAIR_INDEX);
      READ(REPAIRS, REPAIR, REPAIR_INDEX);
      if UNIT.MODEL = MODEL_KEY and REPAIR.DATE >= DATE then
         WRITE(SELECTIONS, REPAIR.DEFECT);
      end if;
      REPAIR_INDEX := REPAIR.NEXT;
   end loop;
   SORT_SELECTIONS(SELECTIONS);
   SUMMARIZE_SELECTIONS(SELECTIONS, SUMMARIES);
   SORT_SUMMARIES(SUMMARIES);
   SET_OUTPUT(REPORT);
   TITLE := {the title};
   NEW_LINE(4); SET_COL(20); PUT(CURRENT_DATE); NEW_LINE(2);
   SET_COL(13); PUT(TITLE); NEW_LINE(60);
   while not END_OF_FILE(SUMMARIES) loop
      READ(SUMMARIES, SUMMARY);
      if LINE > 54 then
         NEW_PAGE; NEW_LINE; PUT("Page: "); PUT(INTEGER(PAGE-1), 3);
         SET_COL(13); PUT(TITLE); SET_COL(61); PUT(CURRENT_DATE); NEW_LINE(2);
         PUT("  # Key    Name"); NEW_LINE(2);
      end if;
      READ(DEFECTS, DEFECT, SUMMARY.KEY);
      PUT(SUMMARY.COUNT, 3); SET_COL(5); PUT(SUMMARY.KEY);
      SET_COL(12); PUT(DEFECT.NAME); NEW_LINE;
   end loop;
   {the summary};
   CLEAN_UP;
exception
   when DEVICE_ERROR | END_ERROR | NAME_ERROR | STATUS_ERROR =>
      CLEAN_UP; PUT("Data Base Inconsistent");
   when others => CLEAN_UP; raise;
end MODEL_DEFECTS_REPORT;
```

```
KBEmacs (ADA) <KBE.DEMO>MODEL_DEFECTS_REPORT.ADA
```

Screen 45

An Efficiency Modification

In Screen 46 the programmer makes an interesting efficiency modification. In doing this he takes advantage of two properties of a chain of repair records. First, since all of the repairs in a chain apply to the same unit, they must all apply to the same model. Second, the repairs in a chain are sorted in order of their dates, with the most recent repair first. Given these two properties it can be seen that if the selection_test fails on a given repair record, then it must fail for all of the rest of the records in the chain since these records apply to the same model and are older. In order to take advantage of this fact, the programmer changes the program so that the enumeration of the repairs in a chain is cut short as soon as the selection_test fails. Under the assumption that the average repair chain is relatively long, and that relatively few repairs will actually be selected, this change leads to a dramatic improvement in efficiency.

It would be nice if KBEmacs had been able to perform the above efficiency transformation itself. However given that this is not the case, it is an important feature of the system that it does not prevent the programmer from making the transformation. In fact, KBEmacs continually displays program code to the programmer precisely so that he can make such changes.

Finishing the Program

In Screen 47, the programmer finishes the program `MODEL_DEFECTS_REPORT` by filling in the title and removing the summary. The full code which results is shown in two parts. Screen 47a shows the declarations in the program. Highlighting is used to show all of the changes since Screen 42a. Screen 47b shows the body of the program. The line_limit is changed from `64` to `65` since zero lines are required for the summary whereas the default assumption used by the constraint function `SIZE_IN_LINES` in the screens above was that the summary would produce one line of output.

direct editing

```
begin
   OPEN(DEFECTS, IN_FILE, DEFECTS_NAME); OPEN(MODELS, IN_FILE, MODELS_NAME);
   OPEN(REPAIRS, IN_FILE, REPAIRS_NAME); OPEN(UNITS, IN_FILE, UNITS_NAME);
   CREATE(REPORT, OUT_FILE, "report.txt"); CREATE(SELECTIONS, INOUT_FILE);
   CREATE(SUMMARIES, INOUT_FILE);
   loop
      begin
         NEW_LINE; PUT("Enter Model Key: ");
         GET(MODEL_KEY);
         READ(MODELS, MODEL, MODEL_KEY);
         exit;
      exception
         when END_ERROR => PUT("Invalid MODEL Key"); NEW_LINE;
      end;
   end loop;
   DATE := QUERY_USER_FOR_DATE("Start of Reporting Period");
   REPAIR_INDEX := NULL_INDEX;
   loop
      while NULL_INDEX(REPAIR_INDEX) and not END_OF_FILE(UNITS) loop
         READ(UNITS, UNIT);
         REPAIR_INDEX := UNIT.REPAIR;
      end loop;
      exit when NULL_INDEX(REPAIR_INDEX);
      READ(REPAIRS, REPAIR, REPAIR_INDEX);
      if UNIT.MODEL = MODEL_KEY and REPAIR.DATE >= DATE then
         WRITE(SELECTIONS, REPAIR.DEFECT);
         REPAIR_INDEX := REPAIR.NEXT;
      else
         REPAIR_INDEX := NULL_INDEX;
      end if;
   end loop;
   SORT_SELECTIONS(SELECTIONS);
   SUMMARIZE_SELECTIONS(SELECTIONS, SUMMARIES);
   SORT_SUMMARIES(SUMMARIES);
   SET_OUTPUT(REPORT);
   TITLE := {the title};
   NEW_LINE(4); SET_COL(20); PUT(CURRENT_DATE); NEW_LINE(2);
   SET_COL(13); PUT(TITLE); NEW_LINE(60);
   while not END_OF_FILE(SUMMARIES) loop
      READ(SUMMARIES, SUMMARY);
      if LINE > 54 then
         NEW_PAGE; NEW_LINE; PUT("Page: "); PUT(INTEGER(PAGE-1), 3);
         SET_COL(13); PUT(TITLE); SET_COL(61); PUT(CURRENT_DATE); NEW_LINE(2);
         PUT("  # Key    Name"); NEW_LINE(2);
      end if;
      READ(DEFECTS, DEFECT, SUMMARY.KEY);
      PUT(SUMMARY.COUNT, 3); SET_COL(5); PUT(SUMMARY.KEY);
      SET_COL(12); PUT(DEFECT.NAME); NEW_LINE;
   end loop;
   {the summary};
   CLEAN_UP;
exception
   when DEVICE_ERROR | END_ERROR | NAME_ERROR | STATUS_ERROR =>
      CLEAN_UP; PUT("Data Base Inconsistent");
   when others => CLEAN_UP; raise;
```

KBEmacs (ADA) <KBE.DEMO>MODEL_DEFECTS_REPORT.ADA Screen 46

```
s-F Fill the title with (MODEL.NAME & " Defects since " & FORMAT_DATE(DATE)). <end>
s-R Remove the summary. <end>
s-<end> Finish editing the procedure MODEL_DEFECTS_REPORT. <end>
c-shift-C Compiling MODEL_DEFECTS_REPORT
```

```
with CALENDAR, FUNCTIONS, DIRECT_IO, MAINTENANCE_FILES, TEXT_IO;
use CALENDAR, FUNCTIONS, MAINTENANCE_FILES, TEXT_IO;
procedure MODEL_DEFECTS_REPORT is
   use DEFECT_IO, MODEL_IO, REPAIR_IO, UNIT_IO, INT_IO;

   package SELECTION_IO is new DIRECT_IO(DEFECT_KEY_TYPE);
   use SELECTION_IO;
   SELECTIONS: SELECTION_IO.FILE_TYPE;

   type SUMMARY_TYPE is
      record
         COUNT: INTEGER;
         KEY: DEFECT_KEY_TYPE;
      end record;
   package SUMMARY_IO is new DIRECT_IO(SUMMARY_TYPE);
   use SUMMARY_IO;
   SUMMARIES: SUMMARY_IO.FILE_TYPE;

   CURRENT_DATE: constant STRING := FORMAT_DATE(CLOCK);
   DEFECT: DEFECT_TYPE;
   DATE: TIME;
   MODEL: MODEL_TYPE;
   MODEL_KEY: MODEL_KEY_TYPE;
   REPAIR: REPAIR_TYPE;
   REPAIR_INDEX: REPAIR_INDEX_TYPE;
   REPORT: TEXT_IO.FILE_TYPE;
   SUMMARY: SUMMARY_TYPE;
   TITLE: STRING(1..41);
   UNIT: UNIT_TYPE;

   function BUILD_SUMMARY(COUNT: INTEGER; KEY: DEFECT_KEY_TYPE)
                          return SUMMARY_TYPE is
      begin return SUMMARY_TYPE'(COUNT, KEY); end BUILD_SUMMARY;
   function SUMMARY_GREATER_THAN(X: SUMMARY_TYPE; Y: SUMMARY_TYPE)
                                 return BOOLEAN is
      begin return X.COUNT > Y.COUNT; end SUMMARY_GREATER_THAN;
   procedure SORT_SELECTIONS is
      new SORT_FILE(DEFECT_KEY_TYPE, SELECTION_IO.FILE_TYPE,
                    SELECTION_IO.POSITIVE_COUNT);
   procedure SORT_SUMMARIES is
      new SORT_FILE(SUMMARY_TYPE, SUMMARY_IO.FILE_TYPE,
                    SUMMARY_IO.POSITIVE_COUNT,SUMMARY_GREATER_THAN);
   procedure SUMMARIZE_SELECTIONS is
      new SUMMARIZE_FILE(DEFECT_KEY_TYPE, SELECTION_IO.FILE_TYPE, SUMMARY_TYPE,
                         SUMMARY_IO.FILE_TYPE, BUILD_SUMMARY);
   procedure CLEAN_UP is
      begin
         SET_OUTPUT(STANDARD_OUTPUT);
         CLOSE(DEFECTS); CLOSE(MODELS); CLOSE(REPAIRS); CLOSE(UNITS);
         CLOSE(REPORT); CLOSE(SELECTIONS); CLOSE(SUMMARIES);
      exception
         when STATUS_ERROR => return;
      end CLEAN_UP;
```

KBEmacs (ADA) <KBE.DEMO>MODEL_DEFECTS_REPORT.ADA

Screen 47a

```
begin
   OPEN(DEFECTS, IN_FILE, DEFECTS_NAME); OPEN(MODELS, IN_FILE, MODELS_NAME);
   OPEN(REPAIRS, IN_FILE, REPAIRS_NAME); OPEN(UNITS, IN_FILE, UNITS_NAME);
   CREATE(REPORT, OUT_FILE, "report.txt"); CREATE(SELECTIONS, INOUT_FILE);
   CREATE(SUMMARIES, INOUT_FILE);
   loop
      begin
         NEW_LINE; PUT("Enter Model Key: ");
         GET(MODEL_KEY);
         READ(MODELS, MODEL, MODEL_KEY);
         exit;
      exception
         when END_ERROR => PUT("Invalid MODEL Key"); NEW_LINE;
      end;
   end loop;
   DATE := QUERY_USER_FOR_DATE("Start of Reporting Period");
   REPAIR_INDEX := NULL_INDEX;
   loop
      while NULL_INDEX(REPAIR_INDEX) and not END_OF_FILE(UNITS) loop
         READ(UNITS, UNIT);
         REPAIR_INDEX := UNIT.REPAIR;
      end loop;
      exit when NULL_INDEX(REPAIR_INDEX);
      READ(REPAIRS, REPAIR, REPAIR_INDEX);
      if UNIT.MODEL = MODEL_KEY and REPAIR.DATE >= DATE then
         WRITE(SELECTIONS, REPAIR.DEFECT);
         REPAIR_INDEX := REPAIR.NEXT;
      else
         REPAIR_INDEX := NULL_INDEX;
      end if;
   end loop;
   SORT_SELECTIONS(SELECTIONS);
   SUMMARIZE_SELECTIONS(SELECTIONS, SUMMARIES);
   SORT_SUMMARIES(SUMMARIES);
   SET_OUTPUT(REPORT);
   TITLE := MODEL.NAME & " Defects since " & FORMAT_DATE(DATE);
   NEW_LINE(4); SET_COL(20); PUT(CURRENT_DATE); NEW_LINE(2);
   SET_COL(13); PUT(TITLE); NEW_LINE(60);
   while not END_OF_FILE(SUMMARIES) loop
      READ(SUMMARIES, SUMMARY);
      if LINE > 65 then
         NEW_PAGE; NEW_LINE; PUT("Page: "); PUT(INTEGER(PAGE-1), 3);
         SET_COL(13); PUT(TITLE); SET_COL(61); PUT(CURRENT_DATE); NEW_LINE(2);
         PUT("  # Key    Name"); NEW_LINE(2);
      end if;
      READ(DEFECTS, DEFECT, SUMMARY.KEY);
      PUT(SUMMARY.COUNT, 3); SET_COL(5); PUT(SUMMARY.KEY);
      SET_COL(12); PUT(DEFECT.NAME); NEW_LINE;
   end loop;
   CLEAN_UP;
exception
   when DEVICE_ERROR | END_ERROR | NAME_ERROR | STATUS_ERROR =>
      CLEAN_UP; PUT("Data Base Inconsistent");
   when others => CLEAN_UP; raise;
end MODEL_DEFECTS_REPORT;
```

KBEmacs (ADA) <KBE.DEMO>MODEL_DEFECTS_REPORT.ADA Screen 47b

Testing

In Screen 48, the programmer tests the program MODEL_DEFECTS_REPORT.

Evaluating the Commands Used

The following summarizes the commands used to construct the program MODEL_DEFECTS_REPORT, replacing most of the direct editing with equivalent "Fill" commands.

```
Define a procedure MODEL_DEFECTS_REPORT.
Insert a simple_report of a file_enumeration of a file_summarization of
  a file_selection of DEFECTS.
Fill the enumerator with an all_chains_enumeration of UNITS and REPAIRS.
Fill the selection_test with
  UNIT.MODEL = {a pre_loop_computation of a query_user_for_key of MODELS}
  and REPAIR.DATE >= {a pre_loop_computation of
                        QUERY_USER_FOR_DATE("Start of Reporting Period")}
Fill the defect_key_accessor with REPAIR.DEFECT.
direct editing -- efficiency transformation.
Fill the title with (MODEL.NAME & " Defects since " & FORMAT_DATE(DATE)).
Remove the summary.
```

Comparison of these commands with the hypothetical directions for a human assistant reproduced below shows that they are qualitatively similar.

```
Define a simple report program MODEL_DEFECTS_REPORT.  Query the user for a
key of a record in the file MODELS and for a starting date.  Enumerate all of
the repairs performed on all of the units and select the repairs which have
been performed on units of the specified model since the start date.  Print a
summarized report showing how many times each kind of defect occurs in these
repairs.  Print a title showing the specified model and the starting date.
Do not print a summary.
```

The key thing to notice about this example is that the program MODEL_DEFECTS_REPORT is quite large. By dint of packing calls on simple I/O procedures several to a line and making use of generic procedures, it is just possible to fit the program in 110 lines. In contrast, the programmer only had to type 11 lines — 7 knowledge-based commands (one of which includes several lines of program text), and the direct editing needed in order to support the efficiency transformation.

```
<select> A
```

```
MODEL_DEFECTS_REPORT;
Enter Model Key: GA2

Enter Date of Start of Reporting Period
Month: 1
Day: 1
Year: 1980

DISPLAY_REPORT;
< 4 newlines>
                    5/27/1985

            Gas Analyzer     Defects since  1/ 1/1980
<59 newlines>
<page>
Page:   1   Gas Analyzer     Defects since  1/ 1/1980        5/27/1985

  # Key     Name

  2 GA-32   Clogged gas injection port
  1 GA-11   Control board cold solder joint
```

Ada Listener

IV - Evaluation

This chapter evaluates KBEmacs in three different ways. It evaluates the pragmatic usefulness of KBEmacs as a program construction tool. It summarizes the principal capabilities of the system and discusses the ways in which KBEmacs is a step in the direction of the PA. It compares KBEmacs with other, similar kinds of programming tools.

KBEmacs as a Program Construction Tool

During program construction, KBEmacs can provide very significant productivity gains. However, as is, KBEmacs is neither fast enough, robust enough, nor complete enough to be a practical tool. In order to fix these problems, the system would have to be reimplemented.

Productivity Gains

The most obvious benefit derived by using KBEmacs is that it makes it possible to construct a program more quickly. The exact increase in speed is difficult to assess. However, one way to make an estimate is to compare the number of commands the programmer has to type in order to construct a program using KBEmacs with the number of lines of code in the program produced. Figure 8 makes this comparison for the various programs and cliches constructed in the scenarios in Chapters II & III. (The cliches PRINT_REPAIR and PRINT_REPAIR_HEADINGS are omitted because they show an anomalously high productivity gain due to the fact that they are largely automatically generated.)

For each program, Figure 8 shows the total number of lines of editing done by the programmer. This is computed as the number of knowledge-based commands which were used plus the number of lines of code which were directly inserted by the programmer. (The total does not include incidental knowledge-based commands such as telling KBEmacs that the program is finished or cosmetic editing requests such as renaming variables.) The total number of lines of editing is then divided into the number of lines of code in the resulting program in order to show the productivity gain.

program	KBEmacs commands	direct editing	total editing	lines of code	productivity gain
REPORT-TIMINGS	5	3	8	19	2
MEAN-AND-DEVIATION	2	4	6	16	3
REPORT-WITH-SUBHEADINGS	3	5	9	37	5
MAINTENANCE_FILES	5	9	14	45	3
UNIT_REPAIR_REPORT	6	0	5	55	9
MODEL_DEFECTS_REPORT	6	5	11	110	10

Figure 8: Productivity gains due to using KBEmacs.

The productivity gains vary from a factor of 2 to a factor of 10. There are four reasons for this variance.

First, programs vary in the extent to which they contain idiosyncratic computation as opposed to computation corresponding to cliches. For example, the program UNIT_REPAIR_REPORT is composed almost exclusively of cliches while the package MAINTENANCE_FILES is composed primarily of idiosyncratic record declarations.

Second, the cliches used in the scenarios differ significantly in how well they take advantage of constraints. For example, when constructing the programs UNIT_REPAIR_REPORT and MODEL_DEFECTS_REPORT constraints automatically specify a substantial number of the cliches used in the programs.

Third, programming languages differ in how many lines of code are required to express a given cliche. In general, Lisp cliches are less than half the size of their Ada counterparts. The difference stems primarily from the fact that Lisp has very concise I/O facilities (e.g., the function FORMAT and the form WITH-OPEN-FILE)

and no data declarations.

Fourth, programs and programming languages differ in how many details they have that KBEmacs can take care of automatically. The productivity gain from taking care of details is most evident in Ada programs where KBEmacs is able to take care of the variable declarations.

At one extreme, productivity gains from using KBEmacs can be very low when constructing a Lisp program such as `REPORT-TIMINGS` which has a fair amount of idiosyncratic structure, does not make particularly good use of constraints, and has only a few details which KBEmacs can take care of automatically. At the other extreme, the productivity gains can be very high when constructing an Ada program such as `MODEL_DEFECTS_REPORT` which has little idiosyncratic structure, makes good use of constraints, and has a lot of details which KBEmacs can take care of automatically.

Although rapid implementation is the most obvious benefit KBEmacs, the increased reliability of the programs produced is at least as important. Both constraints and the use of predefined cliches reduce the number of errors in the programs produced. This reduction of errors is an important (though difficult to quantify) productivity gain in its own right because it saves time in later testing and debugging of the program.

In order to get the full benefit of cliches, the programmer has to make a few compromises. In particular, the programmer has to try to think as much as possible in terms of the cliches available so that cliches can be used as often as possible. Similar kinds of accommodations have to be made in order to fully benefit from a human assistant.

It is important to note that while using cliches is advisable, it is not required. Rather, the benefits of the system gracefully degrade as cliches are used less and less. For example, consider the issue of efficiency. Using cliches seldom leads to a program of optimal efficiency. However, the efficiency of the program can be improved by tailoring the code produced via cliches so that it takes advantage of various special features of the situation. The programmer can get any level of efficiency in the final program he desires depending on how hard he wants to work for it. At any level of efficiency, the program can be constructed faster with KBEmacs than with other methods.

Problems

KBEmacs is a research experiment. Rapid prototyping and rapid evolution have been the only goals of the current implementation. As a result, it is hardly surprising that KBEmacs is neither fast enough, robust enough, nor complete enough to be used as a practical tool.

The most obvious problem with the current implementation of KBEmacs is that it is much too slow. Knowledge-based operations on large programs can take longer than 5 minutes. Even simple operations on small programs take 5-10 seconds. In contrast, experience with interactive systems in general suggests that in order for a system to be judged highly responsive, operations must be performed in at most 1-2 seconds. In order to approach this goal, KBEmacs would have to be speeded up by a factor of 100.

In general, such large speed increases are not easy to obtain. However, in the case of KBEmacs, the required increase seems plausible. There is a lot of room for improvement because, up to now, efficiency has not been a goal of the KBEmacs implementation. As a result, most of the parts of the system are startlingly inefficient. (For example, the Ada parser used by KBEmacs was written in only a few days and is so primitive that it parses only about one line per second.) The last section of Chapter V discusses an experiment which shows how the central operations of KBEmacs can be straightforwardly speeded up by a factor of thirty.

One might comment that it is not sufficient merely to make KBEmacs run acceptably fast on a Lisp Machine, because machines of this power are too expensive to use as programmer work stations. However, if the reduced cost of such computers coupled with the increased cost of programmers has not yet made such computers economical, it soon will. A basic goal of the PA project is to look forward into the future when

computers 10, or even 100, times as powerful as the Lisp Machine will be economical and investigate how this power can be harnessed to assist a programmer.

Another severe problem with the current implementation is that it is fraught with bugs. KBEmacs operates correctly on the scenario shown above. However that scenario amounts to more than half of the testing the system has ever received. In addition, during this testing there has been no visible diminution in the rate at which bugs have been discovered. This suggests that only a small percentage of the bugs to be found have been detected so far.

It appears that it is not practical to attempt to fix the bugs in the system without rewriting it completely. The problem is that, in its role as a continuing research experiment, the system has evolved so extensively and haphazardly beyond the scope of its original design that it is now very difficult to fix a bug without creating another one. The only solution to this is to start again with a new design which is compatible with what the system is now doing.

Another problem with KBEmacs is that it doesn't handle full Ada or even full Lisp. The system handles perhaps 90 percent of Lisp, but no more than 50 percent of Ada. (The specific deficiencies of KBEmacs in this regard are summarized in the next section and discussed in detail in Chapter V.) In order to be a practical tool, KBEmacs would have to be extended to deal with the complete target language in at least a minimal way.

Another problem with KBEmacs as it stands is that the cliche library contains only a few dozen cliches (see Appendix A). Several lines of research (see [Barstow 77; Rich 80; Rich & Waters 81]) suggest that hundreds of cliches are required for an understanding of even a small part of programming and that thousands of cliches would be required for a comprehensive understanding of programming. To a considerable extent there is no barrier to adding the additional cliches which are required. However, designing the exact cliches to use would be a lengthy task. Also, if a large number of cliches were defined, some sort of indexing facility would probably be required in order to help the programmer retrieve cliches.

Fortunately, although the problems above are quite serious, it appears that they could be overcome by reimplementing KBEmacs from scratch with efficiency, robustness, and completeness as primary goals. Unfortunately, since KBEmacs is quite large (some 40,000 lines of Lisp code) reimplementation would be an arduous task. As a result, it would probably be most practical to reimplement the system initially for a small language such as Pascal or basic Lisp rather than for a large language such as Ada.

KBEmacs as a Step Toward the PA

This section compares the capabilities supported by KBEmacs with the capabilities the PA is intended to support. It begins by discussing the depth of understanding of programs exhibited by KBEmacs and then summarizes the capabilities supported by KBEmacs. The next section outlines the additional capabilities which will be supported by the next demonstration system.

The Depth of Understanding of KBEmacs

One of the most fundamental ways to evaluate KBEmacs is to consider what aspects of programming it understands and what aspects it does not understand. To a considerable extent, this boils down to a consideration of what can be represented by the plan formalism. The expressiveness of the plan formalism is summarized below and discussed further in Chapter V.

Via the plan formalism, KBEmacs' understands a variety of programming concepts — in particular data flow, control flow, inputs, outputs, loops, recursion, and subroutine calling. This is enough to form the basis for an understanding of what could be termed *computational algorithms*. However, it is not enough to be a complete understanding of programming. In particular, as will be discussed shortly, it does not include an understanding of either data structures or specifications.

An important part of KBEmacs' understanding of computational algorithms is an understanding of the way in which programming languages represent computational algorithms. This knowledge is procedurally embedded in modules which translate back and forth between the plan formalism and various programming languages (see Chapter V). In particular, KBEmacs has a relatively comprehensive understanding of the language Lisp and an illustrative understanding of the language Ada.

It should be noted that, although KBEmacs' understanding of computational algorithms is fairly extensive, there are several areas where its understanding is either weak or nonexistent. For example, it has no understanding of non-local flow of control such as interrupts. As another example, although KBEmacs has a comprehensive understanding of single-entry loops and simple recursion, it has only a very weak understanding of multiple-entry loops, and multiple and mutual recursion.

A primary limitation of KBEmacs stems from the fact that the plan formalism does not represent any information about data structures. As a result, the small amount of understanding of data structures incorporated into KBEmacs is supported by special purpose procedures. For example, KBEmacs' support for Ada data declarations is wholly procedurally based.

Another key limitation of KBEmacs stems from the fact that the plan formalism does not represent any information about specifications. This makes it very difficult for KBEmacs to reason about whether or not a given algorithm is being used appropriately. (To a certain extent KBEmacs' support for constraints can be used to make up for a lack of understanding of specifications — e.g., by explicitly specifying something that should follow generally from some specification.)

The most important leverage provided by KBEmacs comes from an understanding of cliches. Via the plan formalism and constraints, a wide variety of cliches can be represented. However, the cliches which can be represented are limited in the same ways in which the algorithms which can be represented are limited. In particular, KBEmacs has no understanding of the specifications for various cliches or of cliched data structures.

An important aspect of KBEmacs' understanding of cliches is an understanding of how to perform a variety of operations on them. For example, KBEmacs knows how to instantiate a cliche and integrate it into a program. An additional aspect of KBEmacs' understanding of cliches is that the plan formalism can represent how a program was built up out of cliches. This supports operations such as creating documentation.

Strongly Demonstrated Capabilities

Several of the capabilities which the PA is intended to support are strongly demonstrated by KBEmacs. Saying that a capability is *strongly demonstrated* is intended to imply two things. First, restricted to the domain of program construction, essentially the full capability that the PA is intended to possess is supported. Second, the capability is supported in a general way which demonstrates the efficacy of the key ideas underlying the PA (i.e., the assistant approach, cliches, and plans).

Rapid program construction in terms of cliches — The dominant activity in the scenarios in the last two chapters is the building up of programs by combining algorithmic cliches. The programmer can use an instance of a cliche to fill in a role of another cliche, or insert it at an arbitrary place in a program. In either case, KBEmacs takes care of ensuring that the instance of the cliche is appropriately integrated with the preexisting parts of the program.

User definition of cliches — Simple syntactic extensions to Lisp and Ada are provided so that the programmer can define new cliches as easily as he can define new subroutines. (Defining cliches is also as hard as defining subroutines — i.e., there is nothing at all easy about deciding exactly what cliches should be defined.)

The fact that it is easy for the programmer to define new cliches has two important benefits. First, it

makes it possible to readily extend KBEmacs into new domains of programming. Second, it makes it possible for the programmer to tailor the system so that it fits in better with his particular style of programming.

Editing in terms of algorithmic structure — The knowledge-based commands make it possible for the programmer to operate directly on the algorithmic structure of a program. The essence of this is the ability to refer to the roles of the various cliches used to construct the program even after these roles have been filled in. Commands are provided for filling, removing, and replacing these roles.

Editing in terms of algorithmic structure elevates the level of interaction between the programmer and KBEmacs to a fundamentally higher plane than text-based or syntax-based editing. Such editing makes it possible to state simple algorithmic changes to a program as simple commands even when the algorithmic changes require widespread textual changes in the program. For example, the command "`Insert a count of the element.`" in Screen 13 causes changes to four separate parts of the program `MEAN-AND-DEVIATION`.

As will be discussed in detail in Chapter V, an unfortunate limitation of KBEmacs is that it does not have a full understanding of the algorithmic structure of a program unless the program has been built up from cliches using KBEmacs. If the program was created by ordinary text editing, or if the structure of the program in terms of cliches has been modified by ordinary text editing, then KBEmacs does not have any understanding of the structure of the program in terms of cliches. When this is the case, the programmer can only refer to roles which have not yet been filled in.

Escape to the surrounding environment — At any moment, the programmer can operate on the program being edited with any of the standard Lisp Machine programming tools. In particular, he can freely intermix text-based editing and syntax-based editing with knowledge-based editing. (The high degree to which these modes can be intermixed is illustrated by the fact that, in addition to pausing in the middle of knowledge-based editing to perform text-based editing, one can use the notation "`{a ...}`" to effectively pause in the middle of text-based editing and issue a delayed request for knowledge-based editing.)

Escape to the surrounding environment is of particular importance because it gives the KBEmacs system the property of additivity. New capabilities are added to the programming environment without removing any of the standard capabilities.

When using KBEmacs, the only limitation on escape to the surrounding environment is that, as mentioned above, KBEmacs is not capable of determining the structure of a program in terms of cliches once the program has been modified by text-based or syntax-based editing. Note however, that this leads only to a partial degradation in the ability of KBEmacs to operate on the program — some knowledge-based commands (e.g., replacing a filled role with something new) are no longer supported, but most knowledge-based editing can still continue.

Substantial programming language independence — As will be discussed in detail in Chapter V, the internal operation of KBEmacs is to a large extent language independent. The scenarios above show that the operations supported by KBEmacs are as appropriate for the language Ada as for Lisp.

The phrase "language independent" is only intended to mean that the system can be used in one language as well as in another. It is not intended to mean that the system can translate programs from one language to another. Translation cannot be trivially supported because, as discussed in the beginning of Chapter III, cliches in different programming languages differ semantically as well as syntactically. However, as will be discussed in Chapter VI, cliches and the plan formalism do have a significant amount of leverage on the task of program translation.

Weakly Demonstrated Capabilities

KBEmacs demonstrates a number of capabilities which the PA is intended to support in addition to the ones discussed above. However, these additional capabilities are only weakly demonstrated. Saying that a capability is *weakly demonstrated* is intended to imply two things. First, even restricted to the domain of

program construction, only parts of the capability are supported. Second, the capability is not supported in a general way. However, the weakly demonstrated capabilities are better than mere mockups because, like the strongly demonstrated capabilities, they are based on the key ideas underlying the PA and their implementation illustrates the leverage of these ideas.

A library of cliches — KBEmacs contains a plan library containing several dozen predefined cliches. This example cliche library is weak in two ways. First, as mentioned above, thousands of cliches would be required in order to represent a comprehensive understanding of programming. Second, the library does not support any kind of indexing scheme. The only way that cliches can be referred to is explicitly by name.

Reasoning by means of constraints — KBEmacs uses simple constraint propagation in order to determine some of the consequences of the programmer's design decisions. When defining a cliche the programmer can specify various constraints between the roles. When the cliche is instantiated, these constraints are checked whenever any of the roles are filled or changed. This makes it possible to propagate information from one role to another. As shown in the scenarios above, constraints can be used both to reduce the number of roles which have to be specified by the programmer and enhance the reliability of the programs produced.

As discussed in the beginning of Chapter III, there is a significant weakness in the way constraints are implemented in KBEmacs. Constraints are specified in terms of functions which operate directly on the internal implementation of the plan formalism. In principle, this allows arbitrarily complex constraints to be stated. However, in practice, this significantly limits the constraints which can be stated. The problem is that users are not expected to have any knowledge of the internal implementation of the plan formalism and, in any event, this internal implementation is very cumbersome to deal with.

Taking care of details — The earliest PA proposals [Rich & Shrobe 76,78; Waters 76] suggested that taking care of details was an area where the PA approach had great promise. However, little support for this was provided until quite recently, when KBEmacs was extended to support Ada. The experience with Ada has clearly shown the utility and importance of taking care of details. Consider for example, the way KBEmacs can generate most of the variable declarations and package references in an Ada program and the special purpose support KBEmacs provides for making sure that each data file in the reporting programs created in Chapter III is opened and closed exactly once.

The weakness of KBEmacs' support for taking care of details is two fold. First, this capability is not supported in any kind of uniform way. Rather, several relatively unrelated examples are supported. Second, only a few examples are supported and these examples are not robustly supported.

All of the capabilities mentioned so far pertain primarily to program construction. KBEmacs provides limited support for two additional capabilities which illustrate how the system could be extended into other parts of the programming process.

Program modification — KBEmacs supports two commands (`"Share"` and `"Replace"`) which support program modification. In addition, constraints and editing in terms of algorithmic structure facilitate modification. The scenarios above show the utility of each of these features during program modification. The way the features are supported shows the leverage the plan formalism has on the program modification task. However, KBEmacs' support for program modification can really only be said to have scratched the surface. All that has been done is to support a few simple things which follow straightforwardly from the other operations supported by KBEmacs.

Program documentation — KBEmacs is able to create a simple comment describing a program and it can use highlighting to indicate how various roles are filled in. In a related capability, KBEmacs keeps track of what roles have not yet been filled in and can report this to the programmer. Again, this capability only scratches the surface of what could be done. The goal is to illustrate the fact that the plan formalism represents a significant amount of information which could be useful in producing documentation.

The Next Demonstration System as a Further Step Toward the PA

The next demonstration system will differ from KBEmacs in two crucial ways. First, it will be based on the plan calculus developed by Rich [Rich 80,81] rather than on the simple plan formalism used by KBEmacs. This will give the new system a deeper understanding of programming. Second, the new system will incorporate a fourth key AI idea — general purpose automated deduction. This will make it possible for the new system to support several new kinds of capabilities.

The Depth of Understanding of the Next Demonstration System

As discussed above, there are a number of fundamental aspects of programming of which KBEmacs has essentially no understanding. In fact, it is perhaps surprising that KBEmacs can do what it does when it understands so little. KBEmacs has been developed in a more or less conscious attempt to see how much functionality can be wrung out of the limited understanding the system possesses. The development has been successful both in showing that a great deal can be done and that there are fundamental limits to what is practical given this level of understanding.

A fundamental goal of the next demonstration system is to show that moving to a higher level of understanding will allow for significantly more functionality. The new system will support this higher level of understanding by using the plan calculus. The plan calculus extends the plan formalism by adding in mechanisms for representing four kinds of information which are not represented by the plan formalism.

First, the plan calculus represents data structures and data structure cliches as easily as computational algorithms. This will make it possible to apply all of the basic capabilities of the system (e.g., rapid construction in terms of cliches) to data structures as well as to computational algorithms. (As can be seen in the construction of the package `MAINTENANCE_FILES` in Chapter III, KBEmacs has some abilities in this regard. However, these abilities are essentially simulated by representing data structure operations internally as computational operations.)

Second, the plan calculus makes it possible to represent specifications for data structures, programs, and cliches. This will enable the new system to reason about which cliches are appropriate in a given situation and detect bugs when a cliche is used improperly.

Third, the plan calculus supports several different mechanisms for representing interrelationships between cliches and between cliches and design concepts. One benefit of this is that it will allow design concepts to act as an index into the cliche library — cliches will be referred to by the relevant design concepts rather than by specific names. The value of this stems from the fact that, while there are several thousand important cliches, there are probably only a few hundred important design concepts — large numbers of cliches correspond to the cross products of various sets of design concepts (for example, there is an enumerator for every kind of aggregate data structure). It is the design concepts which form the key vocabulary which the programmer and the system must have a mutual understanding of.

Fourth, in the plan calculus, constraints are represented as predicate calculus expressions. (In KBEmacs, constraints are represented by procedures which are outside of the plan formalism.) Representing constraints as predicate calculus expressions has several advantages. Most importantly, since the constraints are simply predicate calculus expressions, a user can specify a new constraint without having to have any knowledge of the internal implementation of the plan calculus. In addition, the new system will be able to reason about the constraints themselves — KBEmacs is limited to merely running them.

The discussion above presents a number of aspects of programming which KBEmacs does not understand but which the next system will understand. It should be noted that there are numerous aspects of programming which the new system will not understand either. For example, the plan calculus does not remedy the defects in the plan formalism with regard to understanding non-local control flow, multiple-entry

loops, or multiple and mutual recursion.

Generalized Automated Deduction

General purpose automated deduction is best understood in contrast to reasoning performed by special purpose procedures. In a general purpose automated deduction system, not only the facts being reasoned about but also various theorems and other reasoning methods are represented as data objects. Only a few very basic reasoning methods (e.g., reasoning about equality) are built into the system. This makes it possible for a general purpose automated deduction system to reason about a wide range of problems and to flexibly use a wide range of knowledge when doing so. In addition, such a system can be straightforwardly extended by adding new theorems and new kinds of knowledge. In contrast, special purpose reasoning systems typically embed theorems in procedures. Such procedures are fundamentally restricted in that each one solves a narrowly defined problem using a limited amount of knowledge. In order to attack a new problem or use additional knowledge, a new procedure has to be written.

In the new system, general purpose automated deduction will be supported by a reasoning module [Rich 82,85] specially designed to work in conjunction with the plan calculus. The reasoning module will make it possible to significantly improve the support for many of the capabilities of KBEmacs. For example, the reasoning module will give a much firmer foundation for constraint propagation and modification capabilities. It should also make it easier for the system to take care of a wider variety of details for the programmer.

Capabilities To Be Demonstrated By the Next System

The new demonstration system will support all of the capabilities supported by KBEmacs. In addition, the combination of general purpose automated deduction and the plan calculus will make it possible for the new system to demonstrate three key capabilities which are not demonstrated by KBEmacs.

Contradiction detection — The most important use of the reasoning module will be *contradiction detection*. In contrast to automatic verification (which seeks to prove that a program satisfies a complete set of specifications), contradiction detection starts with whatever partial specifications are available for a program and locates bugs in the program by discovering any obvious contradictions.

Data type checking (e.g., as performed by an Ada compiler) is a simple example of contradiction detection. A data type checker locates contradictions based on specifications for the data types of various quantities. The new system will be much more powerful than a type checker because it will be able to utilize a much wider variety of specifications. The system will also be more flexible than current type checkers because it will not require that every contradiction detected be immediately fixed. It is often important to be able to temporarily ignore minor problems so that more serious problems can be investigated (e.g., through testing).

As an example of the utility of contradiction detection, consider the following. Since KBEmacs does not support any contradiction detection, the system has no way to determine whether a set of commands is reasonable. Rather, it just does whatever the programmer says. For example, if the programmer had told KBEmacs to implement the enumerator in the program `UNIT_REPAIR_REPORT` as a vector-enumeration of `REPAIRS`, KBEmacs would have gone right ahead and done it. In contrast, the next demonstration system will be able to complain that such a command does not make any sense.

The major benefit of contradiction detection is that it is largely orthogonal to program testing. The bugs which are easy to find by contradiction detection are often quite different from the bugs which are easy to find by testing. As a result, applying contradiction detection in addition to program testing can significantly increase ones confidence in a program.

Recognition of the cliches which could have been used to construct a program — The PA will be able to recognize what cliches could have been used to construct a program no matter how the program was created. This will make it possible to fully support editing in terms of algorithmic structure at all times. As will be discussed in Chapter VI this also provides a basis for attacking the task of program translation.

Interaction in terms of design decisions — The fundamental level of interaction between a programmer and KBEmacs is in terms of specific algorithms (cliches). An important goal of the next demonstration system is to go beyond this level of interaction and support interaction in terms of design decisions. For example, when constructing the program `UNIT_REPAIR_REPORT`, the programmer had to specify the use of the cliche chain_enumeration. In the new system the programmer will instead merely have to say that the repair records are in a chain file. The system will be able to conclude that in order to print a simple_report of repairs, the cliche chain_enumeration must be used.

Graduating to the level of interaction in terms of design decisions will have a pervasive effect on the capabilities of the system. For example, the programmer will be able to go beyond editing in terms of algorithmic structure and edit in terms of design decisions. As another example, the focus will shift from defining small suites of cliches, to defining large classes of cliches which define the effects of design decisions. In addition, reasoning by propagation of design decisions will be used both as a primary basis for contradiction detection and in order to relieve the programmer from the need to specify things which are implied by other design choices.

The PA

Moving from KBEmacs to the next demonstration system, the emphasis will be on increasing the depth of understanding and therefore the capabilities of the system. In contrast, when moving from the next demonstration system to the PA the emphasis will be on widening the range of applicability of the system beyond program construction.

The next demonstration system will support most of the capabilities which the PA is intended to support. The PA will attempt to apply these capabilities to all of the phases of the programming process. In particular, the PA will focus on the tasks of requirements analysis and modification because these are inherently more important parts of the program life cycle than program construction.

Related Work

This section compares KBEmacs with other approaches to improving programmer productivity. The comparison focuses on tools and projects which are either similar in their capabilities or based on closely related ideas. Projects which share ideas with KBEmacs in that they have intentionally used ideas from KBEmacs (or the PA project as a whole) are discussed in Chapter VI.

KBE

As discussed in Chapter I, KBEmacs is the second in a series of demonstration systems which are heading in the direction of the PA. The first demonstration system was the Knowledge-Based Editor (KBE). KBEmacs is implemented as an extension of KBE, and for the most part, the capabilities of KBE (see [Waters 82a]) are a subset of the capabilities of KBEmacs.

Rapid program construction in terms of cliches and editing in terms of algorithmic structure were the main focus of KBE. In these areas, KBE supported essentially the same capabilities as KBEmacs in essentially the same way. In addition, KBE provided the same weak support for program modification that KBEmacs provides.

The most obvious difference between KBE and KBEmacs is that while KBE was a stand-alone system, KBEmacs is embedded in the standard Lisp Machine Emacs-style program editor. As part of this change, KBEmacs provides much better support for escape to the surrounding environment. (As an additional feature in this direction, KBEmacs introduced the ability to use the notation "`{a ...}`" when text editing.) Although a programmer could apply text-based editing to a program constructed using KBE this could not be done until after all of the roles had been filled in.

KBEmacs supports a number of capabilities which were not supported by KBE at all. For example, KBEmacs supports a syntactic representation for cliches which allows a programmer to easily define new cliches. The most important part of this is the ability to use `{...}` annotation for input from the programmer as well as output to the programmer. In addition, KBEmacs introduces support for constraints and program documentation.

Another advance of KBEmacs is that it introduces support for Ada. (KBE only supported Lisp.) As part of this, KBEmacs provides greatly increased support for taking care of details. The speed and robustness of the system was also increased in order to make it possible to operate on Ada programs of realistic size. (Whereas the largest program ever constructed using KBE was only about 15 lines long, KBEmacs has been used to construct programs more than 100 lines long.)

It is interesting to note that KBE supported a few features which are no longer supported by KBEmacs. For example, KBE was able to draw a diagram showing the plan which corresponded to the program being constructed. It turned out that these diagrams were not very helpful because they were hard for people to read and understand. The basic problem is that while simple diagrams are very easy to understand (perhaps easier than the corresponding program), complicated diagrams are almost impossible to understand (much harder than the corresponding program). As an example of this problem, consider data flow. Explicit data flow arcs have very nice semantics and are very easy to comprehend in isolation. However, if a diagram contains a large number of data flow arcs, then the diagram has to be drawn with extraordinary care in order for the reader to be able to identify and follow the individual arcs (particularly when they cross). In order to render the various arcs legibly, the typical plan diagram requires from 100 to 1000 times the space of the corresponding program text. All in all, experience revealed that program text was a much more useful user interface than plan diagrams.

The argument above should not be taken to imply that graphical output is never useful. People seem to have a great affinity for structure diagrams and the like. However, note that these diagrams achieve simplicity

because an enormous amount of information is left out. In the realm of documentation it would probably be very useful for KBEmacs to be able to create simple diagrams which represent various subsets of the information in a plan.

A subtle, but pervasive, difference between KBEmacs and KBE is that, while both systems are based on the plan formalism, only KBE advertised this fact to the user.

During the design of KBE, the assumption was made that, like the system, the user was thinking in terms of plans. It was considered incidental that program text as opposed to plans was being displayed to the user. This led to the design of a language for knowledge-based commands which was couched in terms of plan concepts.

However, experimentation with KBE revealed that programmers considered it far from incidental that program text was being displayed and that, in most situations, they preferred to talk in terms of the text rather than in terms of the underlying plan. As a result, KBEmacs attempts wherever possible to project the image that it is operating in a primarily textual manner. This has progressed to the point where it is possible to tell someone how to use KBEmacs without mentioning the plan formalism at all.

As an example of the way the interface to KBE evolved, consider the following. When the knowledge-based command language for KBE was first designed, one had to refer to an output port in a program in plan terms — one could not refer to it via the name of a variable carrying the value. For example, if a program contained the form "`(SETQ X (CAR L))`", one was forced to refer to the value in the variable `X` as "`the output of the CAR`". It rapidly became clear that programmers felt that it was very important to be able to refer to ports via variable names and this capability was added into the command language. Simultaneously the ability to refer to ports (other than output roles) in plan terms atrophied away because no one used it.

Programming Environments

KBEmacs is not intended to be a complete programming environment. Rather, it is intended that KBEmacs coexist with a variety of other tools in a standard programming environment. Most programming tools (e.g., version control systems, high level design aids, flowchart drawers, systems for managing the programming process) are orthogonal to KBEmacs in that they provide capabilities which are outside of the scope of the system. KBEmacs neither renders these tools obsolete nor is rendered obsolete by them.

However, research on the PA is relevant to programming environments in general. It is part of a general research trend toward having a central repository containing a large amount of information about an evolving program and having a variety of tools which interact with this information.

Current programming environments (see for example [Dolotta 76; Lisp 84]) allow various programming tools to be flexibly used together. However, the tools only interact loosely — little information other than program code is passed from one tool to another.

In contrast, many researchers (see for example the work on PSL/PSA [Teichroew 77], KBPA [Harandi 83], and KBSA [Green & Rich 83]) are beginning to focus on environments of tightly interacting tools. The PA can be looked at as such an environment where a number of tools interact with a central knowledge base of plans.

Program Editors

KBEmacs is essentially a program editor and every effort has been taken to ensure that programmers can think of it as an additive extension to a state-of-the-art program editor. However, this extension is a very powerful one. As a result, KBEmacs is quite different from other program editors.

A key dimension on which to compare program editors is their level of understanding of the programs being edited (see Figure 9). The level of understanding is important because it determines the kinds of operations an editor can perform.

The simplest program editors are merely ordinary text editors which are used for editing programs. These editors have no understanding of programs at all. The operations supported by these editors are limited to operations on characters — e.g., inserting, deleting, and locating character strings.

PROGRAM EDITOR	LEVEL of UNDERSTANDING
Text Editor	Character Strings
Syntax Editor	Parse Trees
KBEmacs	Algorithmic Structure
Next Demonstration System	Design Decisions
:	:

Figure 9: Levels of understanding exhibited by program editors.

Some program editors incorporate an understanding of the syntactic structure of the program being edited (see for example [Stallman 81; Donzeau-Gouge 75; Medina-Mora 81; Teitelbaum 81]). This makes it possible to support operations based on the parse tree of a program — e.g., inserting, deleting, and moving between nodes in the parse tree.

An important aspect of syntax-based editors is that they can ensure that the program being edited is always syntactically correct. It has been shown [Reps 83] that syntax-based editors can succeed in utilizing essentially all of the information which has traditionally been expressed in terms of syntax (e.g., data type constraints).

KBEmacs goes beyond existing program editors by incorporating an understanding of the algorithmic structure of the program being edited. By means of the plan formalism, KBEmacs understands what cliches were used to construct the program as well as the basic operations, data flow, and control flow in the program. This understanding makes it possible for KBEmacs to support operations based on algorithmic structure — e.g., instantiating cliches and filling roles. In addition, via constraints, KBEmacs can ensure certain aspects of the semantic correctness of the program being edited.

The next demonstration system will take a step further by incorporating an understanding of the design decisions which underly the choice of algorithms in a program. This will make it possible for the programmer to converse with the system in terms of these decisions instead of in terms of specific algorithms. Levels of understanding much higher than those shown in Figure 9 are possible — e.g., understanding the trade-offs which underly design decisions. Reaching these higher levels is a long term goal of the PA project.

Syntax Editors

An unfortunate aspect of syntax-based editors is that (with the notable exception of Emacs [Stallman 81]) early syntax-based editors significantly (if not totally) restricted the programmer's ability to use text-based editing commands. As discussed in detail in [Waters 82b], such restrictions are frustrating in many ways. Syntax-based commands are more convenient than text-based commands in many situations; however, there

is no reason why programmers should be forced to use syntax-based commands when they are not more convenient.

Although most syntax-based editors still do not support text-based commands, at least one recent syntax-based editor (SRE [Budinsky 85]) fully supports them. In keeping with the assistant approach, KBEmacs pays scrupulous attention to fully supporting text-based and syntax-based commands in addition to knowledge-based commands.

An interesting aspect of syntax-based editors is the extent to which they support cliches. Almost every syntax-based editor provides cliches corresponding to the various syntactic constructs in the programming language being used. However, almost no syntax-based editor supports any other kind of cliche. To a certain extent this seems to be a missed opportunity (see the discussion of the Tempest editor [Sterpe 85] in Chapter VI). However, one cannot support more complex cliches in the same way that simple cliches corresponding to syntactic constructs are supported. In particular, in order to be able to flexibly combine complex cliches, an editor must have an understanding of the semantic structure of the cliches. For example, if two cliches use the same variable, then one of the uses of this variable will, more than likely, have to be renamed before the cliches can be combined. The primary advantage of KBEmacs over syntax-based editors is that, due to its understanding of semantic structure, KBEmacs can support the manipulation of complex cliches.

Program Generators

Program generators have the same overall goal as KBEmacs — dramatically reducing the effort required to implement a program. Further, they are in essence similar to KBEmacs in the way in which they reduce programmer effort — they embody knowledge of the cliched aspects of a class of programs. Where program generators are applicable, they are significantly more powerful than KBEmacs. However, program generators are only applicable in certain narrow domains.

In its purest form, a program generator obviates the need to have any programmer at all. An end user describes the desired results in a special problem-oriented language (or through some interactive dialog with the generator) and a program is produced automatically. A key aspect of this is that no one ever looks at the program produced — all maintenance is carried out at the level of the problem-oriented language.

There is ample evidence that a program generator can be created for any sufficiently restricted domain. An area where program generators have been particularly successful is data base management systems. Dozens of program generators in this area exist, perhaps the most successful of which is Focus [Focus 85].

The problem with program generators is their narrowness of scope. Even a tiny bit outside of its scope, a given program generator is nearly useless. In order to get the flexibility they need, programmers sometimes resort to using a program generator to make an approximately correct program and then manually modifying the code produced in order to get the program desired. Unfortunately, there are two major problems with this. First, even the smallest manual change forces further maintenance to be carried out at the level of the code instead of at the level of the problem-oriented language. Second, since the code created by a program generator is usually not intended for human consumption, it is typically quite difficult to understand and therefore modify.

The principal advantage of KBEmacs over program generators is that it has a wide range of applicability. The prime domain of applicability of KBEmacs is defined by the contents of the cliche library. Within this prime domain, KBEmacs is somewhat similar to a program generator though its interface is very different — the key information which has to be specified to KBEmacs in order to create a program is similar to what must be specified to a program generator. Beyond the edge of its prime domain of applicability, KBEmacs is very different from a program generator — it continues to be useful because the programmer can freely intermix cliched and non-cliched computation. Instead of an abrupt reduction in utility, the utility of

KBEmacs is reduced gradually as one moves farther and farther from its prime domain. In addition, the prime domain can be extended by defining new cliches.

The research on program generators is pursuing the goal of increasing the size and the complexity of the domains which can be handled. The PHI-nix system [Barstow 82] is an interesting example of a program generator in a complex domain. Given a description which is primarily composed of mathematical equations, PHI-nix is capable of generating complex mathematical modeling programs in the context of petroleum geology. In order to do this, the system combines a variety of techniques including symbolic manipulation of mathematical expressions.

The CAP system [Bassett 84] uses a few of the same ideas as KBEmacs in order to increase the size of the domain of applicability. In particular, CAP makes use of what is effectively a library of cliches rather than being solely procedurally based. Using CAP, a programmer can build a program by combining *frames* (cliches) which have *breakpoints* (roles) that can contain custom code which becomes part of the generated program. The key weakness of CAP is that it does not have any semantic understanding of its frames and therefore is limited in its ability to manipulate and combine them.

It should be noted that KBEmacs is not really in competition with the program generator approach. Although KBEmacs has some fundamental advantages over program generators, it does not render them obsolete. Similarly, anything short of an all encompassing program generator would not render KBEmacs obsolete. Rather, program generators and KBEmacs can be synergistically combined. Program generators incorporated into KBEmacs can be used to generate special purpose sections of code and KBEmacs can integrate this code into the program being constructed. This makes it possible for the programmer to get the benefit of the program generator without being unduly limited by its narrow focus. (This is illustrated by the constraint function `CORRESPONDING_PRINTING` discussed in Chapter III.)

Transformations

Program transformations have important similarities to the cliches supported by KBEmacs. A program transformation takes a part of a program and replaces it with a new (transformed) part. Typically, a program transformation is *correctness preserving* in that the new part of the program computes the same thing as the old part. The purpose of a transformation is to make the part *better* on some scale (e.g., more efficient or less abstract). For example, a transformation might be used to replace "`X**2`" with "`X*X`".

As usually implemented, transformations have three parts. They have a pattern which matches against a section of a program in order to determine where to apply the transformation. They have a set of applicability conditions which further restrict the places where the transformation can be applied. Finally, they have a (usually procedural) action which creates the new program section based on the old section.

There are two principal kinds of transformations: *vertical* and *lateral*. Vertical transformations define an expression at one level of abstraction in terms of an expression at a lower level of abstraction — for example defining how to enumerate the elements of a vector using a loop. Lateral transformations specify an equivalence between two expressions at a similar level of abstraction — for example specifying the commutativity of addition. Lateral transformations are used principally to promote efficiency and to set things up properly so that vertical transformations can be applied.

The most common use of transformations is as the basis for *transformational implementation systems*. Such systems are typically used to convert programs expressed in a high level non-executable form into a low level executable form. In many ways, a transformational implementation system can be looked at as a special kind of a program generator. The principal difference between transformational implementation systems and program generators is that the knowledge of how to implement programs is represented in a data base of transformations rather than in a procedure. This makes it easier to extend and modify a transformational implementation system; however, it brings up a new problem: controlling the application of transformations.

The difficulty is that, in a typical situation, many different transformations are applicable and different results will be obtained depending on which transformation is chosen. With regard to vertical transformations, the problem is not too severe because usually only a few transformations will be applicable to a given abstract expression. However, the selection of lateral transformations typically has to rely on manual control.

Existing transformational implementation systems can be divided into two classes: those that are relatively limited in power but require no user guidance and those which are capable of very complex implementations but only under user guidance. TAMPR [Boyle 84] and PDS [Cheatham 84] use simple control strategies and restrictions on the kinds of transformations which can be defined in order to obviate the need for user guidance. PDS is particularly interesting due to its emphasis on having the user define new abstract terms and transformations as part of the programming process.

PSI was one of the first systems to use a transformational implementation module for complex implementation. PSI's transformational module [Barstow 77] operated without guidance, generating all possible low level programs. It was assumed that another component [Kant 79] would provide guidance as to which transformations to use. Work on complex transformational implementation systems is proceeding both at the USC Information Sciences Institute [Balzer 81; Wile 82] and the Kestrel Institute [Green 81]. A key focus of both these efforts has been attempting to automate the transformation selection process [Fickas 83].

An interesting system which bridges the gap between transformational implementation systems and more traditional program generators is Draco [Neighbors 84]. Draco is a transformational framework in which it is easy to define special purpose program generators. When using Draco to define a program generator for a domain, a "domain designer" follows the classic program generator approach of designing a problem-oriented language which can be used to conveniently describe programs in the domain and then specifying how to generate programs based on the problem-oriented language. The contribution of Draco is that it provides a set of facilities which make it possible to specify the program generation process in a way which is primarily declarative as opposed to procedural. BNF is used to define the syntax of the problem-oriented language while lateral and vertical transformations are used to define its semantics. Procedures are used only as a last resort. When a program is being implemented by a Draco-based program generator, the user is expected to provide guidance as to which transformations to use.

In their essential knowledge content, KBEmacs' cliches are quite similar to vertical transformations. However, cliches are used differently. Instead of applying correctness preserving transformations to a complete high level description of a program, the user of KBEmacs builds a program essentially from thin air by applying non-correctness preserving transformations. This supports an evolutionary model of programming wherein key decisions about what a program is going to do can be deferred to a later time as opposed to a model where only lower level implementation decisions can be deferred.

A more important difference between cliches and transformations is that cliches operate in the domain of plans rather than program text or parse trees. This raises the level at which knowledge is specified from the syntactic to the semantic. In addition, cliches are completely declarative whereas the action of a transformation is typically procedurally specified. Although KBEmacs does not make much use of this feature, the next demonstration system will. In particular, the new system will be able to reason about the action performed by a cliche in order to reduce the need for user guidance.

Very High Level Languages

The greatest increase in programmer productivity to date has been brought about by the introduction of high level languages. The advantage of high level languages is that they significantly reduce the size of the code which is required to express a given algorithm by allowing a number of low level details (e.g., register allocation) to go unstated. The key to this is the existence of a compiler which takes care of the low level

details.

A logical next step would be the introduction of a very high level language which would provide a further significant reduction in the size of program code by allowing a number of middle level details (e.g., data structure selection) to go unstated. The key to this in turn would be the existence of a very powerful compiler which could take care of these middle level details.

The most persistent problem encountered by developers of very high level languages is that the more general purpose the language is, the harder it is to create a compiler which generates acceptably efficient low level code. This problem exists for high level languages as well. However, the inefficiencies have been reduced to a relatively low level and programmers have, for the most part, learned to live with them.

Program generators can be looked at as compilers which support special purpose very high level languages. Several lines of research are directed toward developing more general purpose very high level languages. For example, Hibol [Ruth 81] and Model [Cheng 84] are very high level languages which are useful in business data processing applications. SETL [Schwartz 75] achieves wide applicability while supporting several very high level operations. Significant progress has recently been made toward compiling SETL efficiently (see [Freudenberger 83]).

Much of the current research on transformations is directed toward the support of very high level languages — in the case of the USC Information Sciences Institute the language Gist [Balzer 81] and in the case of the Kestrel Institute the language V [Green 81]. Neither of these systems is complete. However, they hold the promise of eventually supporting truly general purpose very high level languages.

The most important thing to say about the relationship between KBEmacs and general purpose very high level languages is that there is no competition between the approaches. Rather, they are mutually reinforcing. As soon as a general purpose very high level language is developed, it can be used as the target language of KBEmacs. The net productivity gain will be the product of the gains due to the language and due to KBEmacs. The basic claim here is that no matter how high level a language is, there will be cliches in the way it is used and therefore KBEmacs can be usefully applied to it.

Intelligent Design Aids

The basic approach taken by KBEmacs transcends the domain of programming. KBEmacs is an example of a general class of AI systems which could be termed *intelligent design aids*. These programs are expert systems in the sense that they attempt to duplicate expert behavior. However, they are quite different from the more familiar kinds of expert systems which perform diagnostic tasks.

A diagnostic expert system typically takes a relatively small set of input data and derives a relatively simple answer. For example, it might take a set of symptoms and derive the name of a disease. In order to derive the answer, the system performs relatively complex deductions utilizing a knowledge base of rules. The emphasis in such a system is on how to control the operation of the rules so that the required deductions can be reliably and efficiently performed without human intervention.

In an intelligent design aid the various components are the same but the emphasis is quite different. In particular, instead of generating a simple answer, a design aid generates a very complex answer — a complete design for the desired artifact. Just representing this design is, of necessity, an important focus of the system. The knowledge base of rules contains various pieces of knowledge about how to design things. However, due both to gaps in this knowledge and lack of adequate control strategies, design aids are not capable of very much automatic design. As a result, the major emphasis of a design aid is on how the user can control the selection of rules and provide specific aspects of the design when the rules are inadequate.

A good example of intelligent design aids outside of the domain of programming are VLSI design systems. Comparison of [Sussman 79] and [Rich & Waters 79] shows that the key ideas of the assistant approach, cliches, and plans can be used to support VLSI design in basically the same way that they can be used to

support programming. For example, the Vexed VLSI design system [Mitchell 85] is startlingly similar to KBEmacs. The interface is different and the domain is of course very different, however the basic capabilities are very much the same. The dominant activity is building up a circuit by means of implementation rules (cliches) chosen from an extendable library. While this is going on, Vexed automatically takes care of a number of details centering around combining the rules correctly. In addition, the user is always free to modify the circuit directly.

Reusable Components

The reuse of components is a common theme which underlies KBEmacs and almost every approach to increasing programmer productivity [Reuse 84]. There are many different ways in which components can be represented — e.g., cliches, subroutines, transformations, and procedurally in compilers and program generators. However, in all these cases, the goal is the same. Essentially the only way to significantly reduce the time it takes to design something is to take advantage of predesigned components.

As discussed in detail in [Rich & Waters 83], KBEmacs benefits from the fact that the cliches expressed in the plan formalism are a particularly good representation for components. Plan cliches are capable of representing a very wide range of components. Their expressiveness is particularly enhanced by the flexibility introduced by roles and constraints. (Only the procedural representation of components is more expressive.) Plan cliches have very convenient combination properties. They can be combined together as easily (and understandably) as subroutine calls. Plan cliches are represented in a declarative way. This makes it easy for an automatic system to manipulate them and reason about them. Finally, plan cliches are inherently programming language independent. This facilitates the construction of programming tools which are programming language independent.

V - Implementation

Figure 10 shows the architecture of the KBEmacs system. KBEmacs maintains two representations for the program being worked on: program text and a plan. At any moment, the programmer can either modify the text or the plan. If the text is modified, the analyzer module is used to create a new plan. If the plan is modified, the coder module is used to create new program text.

In order to modify the program text, the programmer can use the standard Emacs-style Lisp Machine editor. In order to modify the plan, the programmer can use the knowledge-based editor implemented as part of KBEmacs. An interface unifies ordinary program editing and knowledge-based editing so that they can both be conveniently accessed through Emacs.

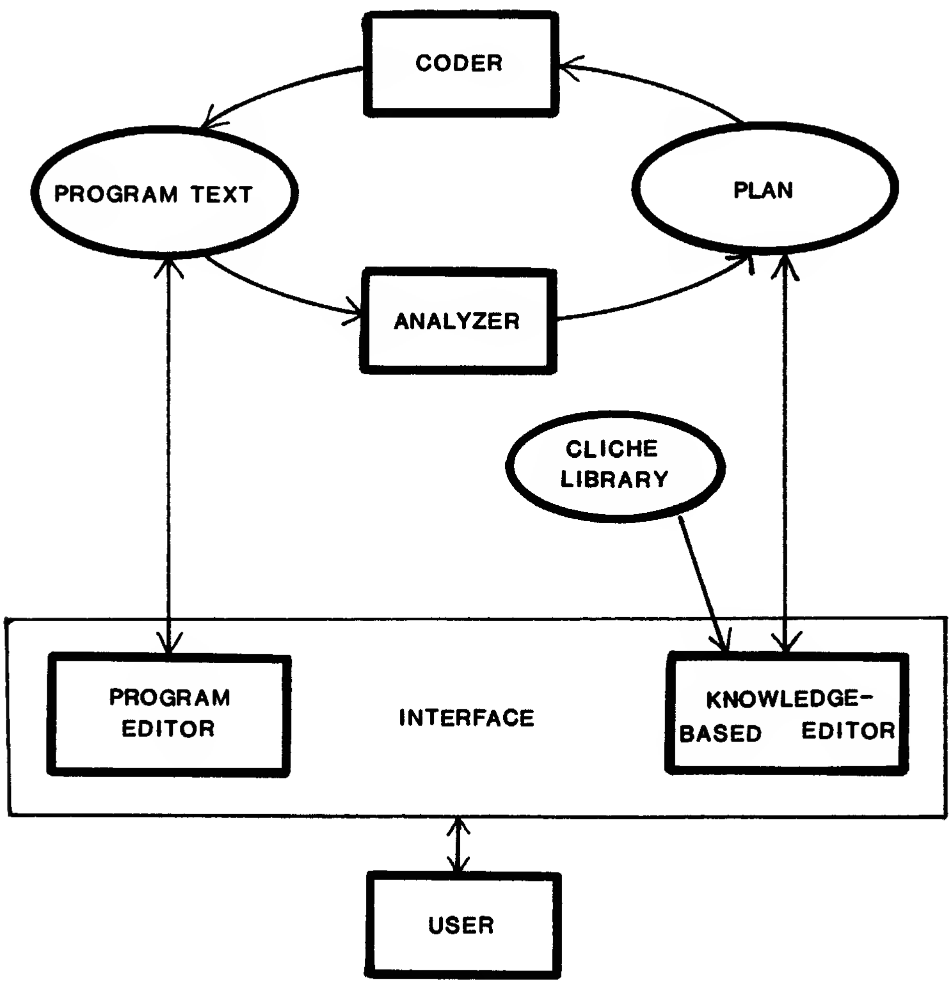

Figure 10: Architecture of KBEmacs.

The sections below begin by discussing cliches and the plan formalism. They then describe how each of the components in Figure 10 is implemented. The discussion focuses on requirements and high level design, attempting to avoid implementation details whenever possible. The discussion also presents some of the ways the components could be improved. In particular, the last section of the chapter presents an experiment which shows how the basic operations of KBEmacs could be speeded up by at at least an order of magnitude.

Cliches

There are five aspects to the way KBEmacs implements cliches. Syntactic extensions of the languages Lisp and Ada provide a textual representation for roles. Additional syntactic extensions enable cliches to be defined in much the same way that subroutines are defined. A simple library mechanism is provided along with a set of example cliches. The plan formalism has a number of features which are provided in order to represent cliches and the way a program is built up out of cliches. Finally, the knowledge-based editor supports a number of commands which make it possible for a programmer to use cliches when editing a program.

The first three aspects of KBEmacs' implementation of cliches are discussed in this section. Appendix A presents all of the cliches currently defined. The plan formalism and the knowledge-based editor are discussed in separate sections below.

Roles And Other Machine Understandable Annotation Of Lisp Programs

As discussed in the beginning of Chapter II, the notation `{...}` is used to represent roles and other machine understandable annotation of Lisp programs. Figure 11 presents the various features of this notation in terms of a simple set of grammar rules. (These rules are expressed in a more or less standard version of BNF which is summarized in a key at the bottom of the figure.)

```
function-call ::= (name {argument}) | role-form | embedded-instance

role-form ::= anonymous-role
            | empty-role
            | (empty-role {argument})
            | '{'form, role'}'
            | ('{'name, role'}' {argument})
anonymous-role ::= '{}'
empty-role ::= '{'role'}'
role ::= the [input | output] name {of the name}
name ::= a non-NIL Lisp symbol

argument ::= form | '{'form, modified'}' | '{'form, depending on role'}'
form ::= function-call | any other Lisp form

embedded-instance ::= '{'a-or-an cliche-name [of instances]'}'
                    | '{'a use of function-reference'}'

        bold-italic-names are non-terminal grammar symbols
        ::= specifies a grammar rule
        ... | ... signifies alternation of possibilities
        [...] signifies optionality
        {...} signifies repetition zero or more times
        's' signifies the terminal grammar string s
        all other symbols are terminal grammar symbols
```

Figure 11: Machine understandable Lisp annotation.

In Lisp code, a role form signifying a role can appear anywhere a function call can appear — i.e., any place where a form will be evaluated (e.g., not as a bound variable or as an entire `COND` clause or inside of a quoted list). As illustrated below there are five kinds of role-forms.

```
an anonymous role                  {}
an empty role                      {input x}
an empty role with arguments       ({operation} A (CAR L))
a filled role                      {(LIST A (CAR L)), operation}
alternate syntax for a filled role ({LIST, operation} A (CAR L))
```

An anonymous role is used as a place holder for something which must be present but which is not important enough to be given a name. Typically it stands in place of something which will be automatically generated by KBEmacs.

An empty role specifies the location of a named role. It can be specified with or without a list of arguments. If a list of arguments is specified then it is expected that the computation which eventually fills the role will use these arguments. The arguments are specified both as a form of documentation for the user and so that KBEmacs can correctly analyze the data flow in the program when the role is not yet filled in.

Usually, roles are not displayed when they are filled in. However, if a filled role is displayed, then the contents of the role precedes the name of the role and is separated from the name by a comma. If a role is filled with a function call (as opposed to some other form such as a constant) then the filled role can be displayed in an alternate way which is sometimes more readable. As illustrated above, the function name is shown as filling the role and the role is shown as having arguments. Note that although the last two examples above are syntactically different they are semantically identical, specifying exactly the same role and its contents.

The name of a role specifies three different things. It specifies the name of the role itself. It specifies whether the role is to be considered to be logically an input or an output of the containing cliche. In addition, it specifies whether the role is part of a larger compound role.

Input and output roles are illustrated in the definition of the cliche list-enumeration shown below. The role list is an input. The role element is an output. The other roles correspond to internal computations and are neither inputs nor outputs. The definition of the cliche list-enumeration also illustrates various kinds of filled and empty role forms. Note that the roles element-accessor and element are filled with the same computation.

```
(DEFINE-CLICHE LIST-ENUMERATION
    (PRIMARY-ROLES (LIST)
     DESCRIBED-ROLES (LIST)
     COMMENT "enumerates the elements of {the list}")
  (LET* ((LIST {the input list}))
    (LOOP DO
      (IF ({NULL, the empty-test} LIST) (RETURN))
      {({CAR, the element-accessor} LIST), the output element}
      (SETQ LIST ({CDR, the step} LIST)))))
```

Compound roles are illustrated in the following excerpt from the cliche simple-report. The empty-test, element-accessor, and step are all part of a compound role, the enumerator.

```
(LOOP DO
  (IF ({the empty-test of the enumerator} DATA) (RETURN))
  (WHEN (> LINE {the line-limit})
    (SETQ PAGE (+ PAGE 1))
    (FORMAT REPORT "~/|~%Page:~3D~50: <~A~>~17A~2%" PAGE TITLE DATE)
    (SETQ LINE 3)
    ({the column-headings} {REPORT, modified} {LINE, modified}))
  ({the print-item} {REPORT, modified}
                    {LINE, modified}
                    ({the element-accessor of the enumerator} DATA))
  (SETQ DATA ({the step of the enumerator} DATA)))
```

Referring back to Figure 11, annotation on the arguments of a function call is a second kind of machine understandable Lisp annotation. There are two types of argument annotation. The first type specifies that the argument is modified (side-effected) by the function it is an argument to. This is illustrated in the excerpt from the cliche simple-report shown above. The column-headings and print-item roles both side-effect their REPORT and LINE arguments. Annotation about side-effects is important so that KBEmacs can correctly

analyze the data flow in a program when a role is not yet filled in.

The second kind of argument annotation has a very specialized meaning. It specifies the logical existence of a role whose value is not actually used for anything. This is illustrated in the cliche count shown below. The cliche count takes in a series of values (e.g., created by enumerating a list) and counts them. The cliche is peculiar in that it depends on the existence of the elements in this series however, it does not actually use any of these elements in its computation. The ramifications of this are discussed in more detail as part of the discussion of the cliche count in Appendix A.

```
(DEFINE-CLICHE COUNT
    (PRIMARY-ROLES (ITEM)
     DESCRIBED-ROLES (ITEM)
     COMMENT "accumulates a count of {the item}")
  (LET* ((COUNT 0))
    (LOOP DO
      (SETQ COUNT (+ COUNT {1, depending on the input item})))
    COUNT))
```

Figure 11 defines a third kind of machine understandable Lisp annotation — an embedded instance. This kind of annotation makes it possible to directly specify a part of a program as an instance of a cliche (e.g., `(+ Y {a squaring of Y})`) or a reference to the return value of a function (e.g., `(+ Y {a use of the CAR})`). Several examples of these are shown in Chapter II as part of the construction of the program `MEAN-AND-DEVIATION`.

When they are used, embedded instances are converted into equivalent program code by the knowledge-based editor module. The exact syntax and meaning of the two forms of an embedded instance is discussed in the section on the knowledge-based editor below.

Defining Lisp Cliches

KBEmacs provides a Lisp macro `DEFINE-CLICHE` for defining cliches. Figure 12 specifies the form such a definition must take. (Note that several of the non-terminals in this figure are defined in Figure 11.)

```
cliche-definition ::= (DEFINE-CLICHE name
                          ({cliche-declaration})
                        {form})
cliche-declaration ::= PRIMARY-ROLES ({name})
                     | DESCRIBED-ROLES ({name})
                     | COMMENT description-string
                     | CONSTRAINTS ({constraint})
description-string ::= "{name | empty-role}"
constraint ::= (DEFAULT empty-role form)
             | (DERIVED empty-role form)
             | (RENAME string form)
string ::= a string literal
```

Figure 12: Defining a Lisp cliche.

As an example of a cliche definition, consider the cliche equality-within-epsilon shown below.

```
(DEFINE-CLICHE EQUALITY-WITHIN-EPSILON
    (PRIMARY-ROLES (X Y)
     DESCRIBED-ROLES (X Y)
     COMMENT "determines whether {the x} and {the y}
              differ by less than {the epsilon}"
     CONSTRAINTS ((DEFAULT {the epsilon} 0.00001)))
  (< (ABS (- {the input x} {the input y})) {the epsilon}))
```

The body of a cliche definition is ordinary lisp code typically containing role annotation. Additional information about the cliche is provided by four kinds of declarations. The `PRIMARY-ROLES` declaration specifies which roles can be directly specified when creating an instance of the cliche. For example, here the programmer can say "`an equality-within-epsilon of A and B`" but he cannot say "`an equality-within-epsilon of A, B, and 0.01`".

The `DESCRIBED-ROLES` and `COMMENT` declarations specify how to create long and short comments describing instances of the cliche. Note that the description string typically contains empty role forms. Although these forms are syntactically identical to empty role forms in the body of a cliche they have a different meaning. When the description string is used, all of the empty role forms in it are replaced by descriptions of the forms which fill the corresponding roles in the body of the instance of the cliche. The use of the `DESCRIBED-ROLES` and `COMMENT` declarations will be discussed when comments are discussed in the section on the knowledge-based editor below.

Constraints

The `CONSTRAINTS` declaration specifies constraints which are placed on the roles. There are three kinds of constraints: default, derived, and rename. The cliche equality-within-epsilon includes a simple example of a default constraint. All three kinds of constraints are illustrated below.

```
(CONSTRAINTS
  ((DEFAULT {the report-file-name} "report.txt")
   (DERIVED {the line-limit} (- 65 (SIZE-IN-LINES {the print-item})))
   (RENAME "RECORD" (CORRESPONDING-RECORD-NAME {the data-file-name}))))
```

Default and derived constraints are very similar. They both take as arguments a role and an expression. The expression is evaluated and used to fill the indicated role. The only difference between the two is that default constraints can be overridden by replacing the contents of the constrained role with some other value. In contrast, if the contents of a derived role are changed, the constraint will act to restore the contents to their original value.

Rename constraints are textual in nature. They take a target string, and an expression which should evaluate to a replacement string. Everywhere in the body of the cliche the target string is replaced by the replacement string. This causes renaming of variable names and function names. It also changes the values of string literals and other literal values. As shown below, the renaming applies not only to the exact target string, but also to compound forms of the target string. In particular, the plural form of the target string is replaced by the plural form of the replacement string and renaming is applied to hyphenated words containing the target string. However, renaming is not applied to names which merely contain the target string as a substring.

```
If "RECORD" is renamed to "UNIT" then

RECORD              becomes     UNIT
RECORDS             becomes     UNITS
RECORD-TYPE         becomes     UNIT-TYPE
"The RECORD key"    becomes     "The UNIT key"
RECORDING           remains     RECORDING
```

There are two key aspects to the way constraints are implemented by KBEmacs. First, a constraint checking module which is part of the knowledge-based editor reevaluates the constraints associated with a cliche whenever the contents of any of the roles in an instance of that cliche are modified. This guarantees that the constraints are checked whenever their effects could be altered.

Second, constraint expressions are composed of ordinary Lisp functions. They are able to affect the roles in an instance of a cliche because they can refer to them by means of empty role forms. The empty role forms

in a constraint expression are syntactically identical to empty role forms in the body of a cliche, however, they have a different meaning. When a constraint expression is evaluated, each empty role form in it evaluates into a direct pointer to whatever fills the corresponding role in the plan for the instance of the cliche the constraint is being applied to. Using this pointer, the constraint expression can inspect or alter the contents (if any) of the role in the plan.

In principle, there is no limit to what can be specified in a constraint expression since the full power of Lisp can be used in these expressions. For example, the constraint function `CORRESPONDING_PRINTING` discussed in Chapter III contains an entire program generator. In practice, constraints are severely limited by the fact that no ordinary user is expected to write a function which can do anything useful with a direct pointer into a plan. As a result, constraints are typically composed of predefined constraint functions. Only a very few constraint functions are defined at the current time.

It should be noted that there are a number of problems with the way the rename constraint is currently supported. In particular, rename constraints are only evaluated once, at the moment the cliche is first instantiated. As a result, any roles which the rename constraint refers to (e.g., the role data-file-name in the example above) must have values specified for them in the knowledge-based command which causes the instantiation of the cliche. This is an unreasonable restriction which could and should be eliminated.

Expressiveness

It is important to consider exactly which algorithms can be expressed as a cliche and which cannot be. Unfortunately, this topic has not been investigated in very much detail (see [Rich & Waters 83] for a brief discussion). However, several points are clear. First, the cliches used in the scenarios in Chapters II & III show that a wide range of cliches can be defined. In particular, many algorithms can be defined which are too fragmentary to be represented as subroutines (e.g., the cliche list-enumeration). Second, both constraints and compound roles greatly extend the expressiveness of cliches.

As an example of the limits to the expressiveness of cliches consider the following. As defined in Chapter I, it is the intention of a cliche that the matrix be non-varying — all of the variability is confined to what fills the roles. (The only exception to this at the current time is the rename constraint.) To a considerable extent this is not a limit because any part of the matrix can be converted into a role. However, no matter how many roles there are, the number of roles and the way they interact is fixed. This is analogous to the situation in many programming languages where the number of arguments to a subroutine is fixed.

Unfortunately, there are algorithms which do not have a fixed number of roles. For example, in KBEmacs, it is easy to define a cliche which computes the average of two numbers or a different cliche which computes the average of three numbers. However, it is not possible to define the cliche which is the generalization of these two cliches (i.e., a cliche which computes the average of any number of numbers) because this cliche has a variable (in fact, unbounded) number of input roles.

Another limitation on cliches in KBEmacs is that since plans cannot represent information about data structures, it is not possible to define cliches which correspond to generalized data types. In Lisp, this restriction is not too obtrusive since the typical Lisp program does not contain any code which describes data types. (As discussed in Chapter IV data cliches will be supported by the next demonstration system.)

The Cliche Library

In KBEmacs, the cliche library is implemented very simply. Property lists are used to associate the plan for a cliche with the name of the cliche. This allows cliches to be rapidly referenced by their names. No other kind of reference is supported.

The analyzer module is used to convert a cliche definition form into a plan for the cliche. The way this is done will be discussed in the section on the analyzer below.

Ada Cliches

Ada cliches are semantically identical to Lisp cliches. Syntactically, they differ in the same way that Ada syntax differs from Lisp syntax. The syntax of machine understandable annotation in Ada is shown in Figure 13. Comparison of this figure with Figure 11 (which defines several of the non-terminal symbols used in Figure 13) shows that there are only two basic differences between Ada annotation and Lisp annotation.

```
ada-function-call ::= name[(ada-argument {, ada-argument})]
                    | ada-role-form | embedded-instance
ada-role-form ::= anonymous-role
                | empty-role[(ada-argument {, ada-argument})]
                | `{`ada-expression, role`}`
                | `{`name, role`}`[(ada-argument {, ada-argument})]
ada-argument ::= ada-expression
               | `{`ada-expression, modified`}`
               | `{`ada-expression, depending on role`}`
ada-field-selection ::= name-or-role.name-or-role{.name-or-role}
name-or-role ::= name | ada-role-form
ada-expression ::= ada-function-call | ada-field-selection
                 | any other Ada expression
ada-statement ::= ada-role-form | an Ada statement
ada-declaration ::= ada-role-form | an Ada declaration
ada-type ::= ada-role-form | an Ada type
```

Figure 13: Machine understandable Ada annotation.

The first difference is that Ada role forms are changed so that they have the same syntax as Ada function calls. For example, as shown below, an empty role with arguments in Ada has the form `{operation}(A, CAR(L))` while in Lisp it has the form `({operation} A, CAR(L))`.

```
an anonymous role                  {}
an empty role                      {input x}
an empty role with arguments       {operation}(A, CAR(L))
a filled role                      {LIST(A, CAR(L)), operation}
alternate syntax for a filled role {LIST, operation}(A, CAR(L))
```

The second difference is that a number of grammar rules have to be modified in order to show all the places where a role form can be used. For example, Ada uses a special syntax for statements (as opposed to expressions) and for selecting the field of a record, where Lisp uses function calls for everything.

It should be noted that Figure 13 shows that roles can appear inside data declarations and types. In Ada, KBEmacs provides simulated support for data cliches (e.g., chain_file_definition) by converting Ada type declarations into an internal form which appears to the system to be executable computation. This is a serviceable but ad hoc approach.

Figure 14 shows the syntax of Ada cliche definitions. Comparison of this figure with Figure 12 reveals that Ada cliche definitions are identical to Lisp cliche definitions except for syntax.

```
ada-cliche-definition ::= cliche name is
                             {ada-cliche-declaration;}
                             {ada-declaration;}
                          begin
                             {ada-statement;}
                          end name;

ada-cliche-declaration ::= primary roles name {, name}
                          | described roles name {, name}
                          | comment description-string
                          | constraints {ada-constraint;} end constraints

ada-constraint ::= DEFAULT(empty-role, ada-expression)
                 | DERIVED(empty-role, ada-expression)
                 | RENAME(string, ada-expression)
```

Figure 14: Defining an Ada cliche.

As illustrated in the example below, the syntax of the cliche defining form is changed so that it is analogous to the way Ada subroutines are defined. (Note that the name of cliche is changed to conform to Ada restrictions on what is a syntactically valid name.)

```
cliche EQUALITY_WITHIN_EPSILON is
   primary roles X, Y;
   described roles X, Y, EPSILON;
   comment "determines whether {the x} and {the y}
            differ by less than {the epsilon}";
   constraints
      DEFAULT({the epsilon}, 0.00001);
   end constraints;

begin
   return abs({the input x} - {the input y}) < {the epsilon};
end EQUALITY_WITHIN_EPSILON;
```

The various cliche declarations are rendered as keywords in analogy with other Ada declarations. Constraints are rendered as Ada function calls. They are implemented by calling the corresponding Lisp constraint functions.

Language Independence

Comparison of the way Lisp and Ada cliches are defined shows the language independence of the ideas involved. It is easy to extend the syntax of almost any language in order to support cliches. Role annotation is supported using the notation `{...}` in analogy with the way function calls are rendered in the language's syntax. Similarly, cliche definition is rendered in analogy with subroutine definition.

In order to support a new language, one also has to provide a library of cliches which are appropriate for writing programs in that language. A separate library of cliches is required because, as discussed in the beginning of Chapter III, programming languages differ semantically as well as syntactically. (Some cliches, particularly very abstract ones, are semantically language independent and therefore could be represented in a language independent cliche library.)

Plan Formalism

The plan formalism is the "mental language" of KBEmacs. As such, it is the most important part of the system. The formalism was originally developed by Rich and Shrobe [Rich & Shrobe 76] and then extended by the author [Waters 76,78]. The subsections below describe the information content of the plan formalism in detail.

Surface Plans

A plan is a set of facts. The facts describe the operations in a program and the data flow and control flow which connect them. Figure 15 shows the types of facts which comprise a *surface* plan. A surface plan describes the basic features of an algorithm without imposing any structure on the algorithm. Figure 16 shows a diagram of a simple surface plan. (This figure is a subset of Figure 3 in Chapter 1 and uses the same diagrammatic conventions -- solid lines represent data flow, and dashed lines represent control flow.) Figure 17 shows the set of facts which correspond to the plan diagram in Figure 16.

```
plan ::= ({fact})
fact ::= seg-description | port | port-description | connection
seg-description ::= (segcase SEG segkind name-or-constant ({form}))
                  | (segcase SEGTYPE segtype)
                  | (segcase WHEN form)
segcase ::= (segment [case])
segment ::= name
case ::= name
name-or-constant ::= NIL | name | Lisp constant
segkind ::= FUNCALL | CONSTANT | LAMBDA-EXP | OPENCODE | RECURSION | JOIN
segtype ::= SIMPLE | SPLIT | JOIN

port ::= (segcase INPUT port-kind obj id)
       | (segcase OUTPUT port-kind obj id)
obj ::= name
id ::= NIL | obj
port-kind ::= NIL | ID | PART

port-description ::= (port IO io-kind name-or-nil argno)
                   | (port CREATEVAR name)
io-kind ::= LAMBDA-ARG | ACTUAL-ARG | FREE-VAR | RETURNVAL | SIDE-EFFECT
name-or-nil ::= name | NIL
argno ::= NIL | an integer

connection ::= (segcase SUBSEG subseg)
             | (segcase RECURSIVE subseg)
             | (segcase DFLOW dflow-from dflow-to)
             | (segcase CFLOW cflow-from cflow-to)
subseg ::= segcase
dflow-from ::= port
dflow-to ::= port
cflow-from ::= segcase
cflow-to ::= segcase
```

Figure 15: The information in a surface plan.

Each fact is a tuple. The second element of the tuple is a keyword which identifies the relationship specified by the fact. The other elements of the tuple are the arguments of the relationship.

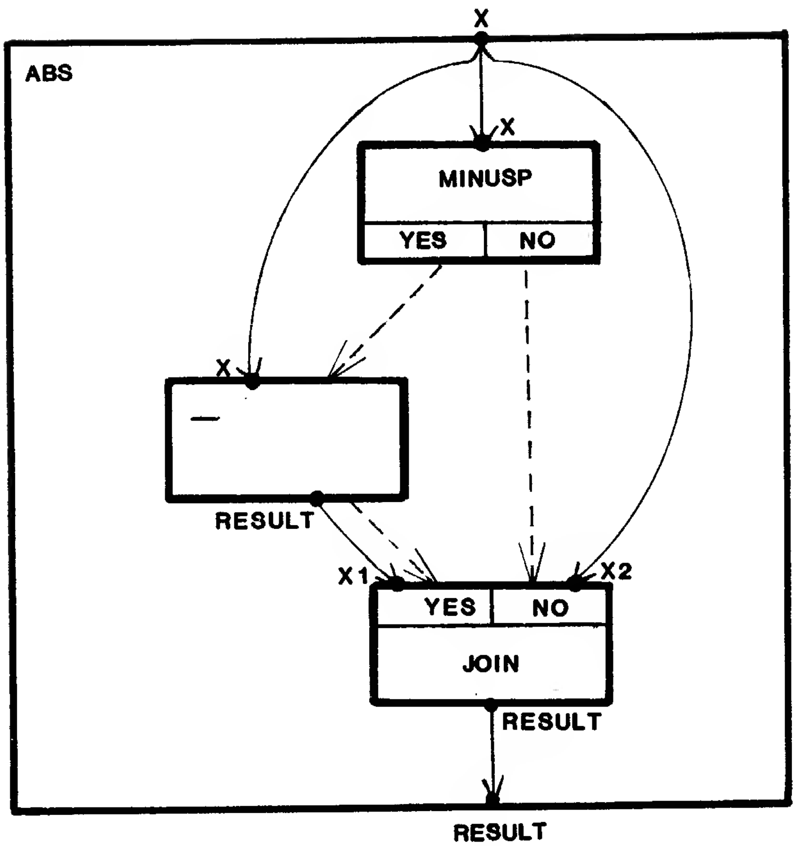

Figure 16: Plan diagram for (COND ((MINUSP X) (- X)) (T X)).

The most basic object in a plan is a *segment*. Segments correspond to a unit of computation. Segments are identified by their names. Each segment in a plan must have a unique name. In facts, segments are referred to by *segcases*. A segcase is a pair of a segment name and an optional *case* name. As will be discussed below, cases are used to identify various control environments associated with a segment. If a fact applies to only one case, then the case is identified. If the fact applies to every case then no case is specified.

There are four different classes of facts in a surface plan. The first class consists of three different facts which describe the basic features of a segment.

A SEG fact specifies three pieces of information about a segment. First it says what kind of segment it is. FUNCALL specifies that the segment is a call on a function. CONSTANT specifies that the segment is a constant such as a number or a string literal. LAMBDA-EXP specifies that the segment corresponds to the definition of a function. OPENCODE specifies that the segment contains a group of other segments, but does not correspond to an entire function definition. RECURSION specifies that the segment is a recursive instance of a containing segment. JOIN specifies that the segment is a *join*. The last two concepts will be discussed in more detail below.

If a segment is a FUNCALL then the SEG fact specifies the name of the function called. If the segment is a CONSTANT then the SEG fact specifies the value of the constant. If the segment is a LAMBDA-EXP then the SEG fact specifies the name of the function defined.

Finally, if possible, a SEG fact specifies what program code corresponds to the segment. The code is specified by using direct pointers into the code for the program so that KBEmacs can keep track of what parts of a plan correspond to what parts of a program.

```
(((MINUSP) SEG FUNCALL MINUSP ((MINUSP X)))
 ((MINUSP) SEGTYPE SPLIT)
 ((MINUSP YES) WHEN (< X 0))
 ((MINUSP NO) WHEN (NOT (< X 0)))
 ((MINUSP) INPUT NIL X NIL)
 (((MINUSP) INPUT NIL X NIL) IO ACTUAL-ARG X 1)

 ((-) SEG FUNCALL - ((- X)))
 ((-) SEGTYPE SIMPLE)
 ((-) INPUT NIL X NIL)
 ((-) OUTPUT NIL RESULT NIL)
 (((-) INPUT NIL X NIL) IO ACTUAL-ARG X 1)
 (((-) OUTPUT NIL RESULT NIL) IO RETURNVAL NIL NIL)

 ((JOIN) SEG JOIN NIL NIL)
 ((JOIN) SEGTYPE JOIN)
 ((JOIN YES) INPUT ID X1 RESULT)
 ((JOIN NO) INPUT ID X2 RESULT)
 ((JOIN) OUTPUT NIL RESULT NIL)
 (((JOIN YES) INPUT ID X1 RESULT) IO FREE-VAR X NIL)
 (((JOIN NO) INPUT ID X2 RESULT) IO FREE-VAR X NIL)
 (((JOIN) OUTPUT NIL RESULT NIL) IO RETURNVAL NIL NIL)

 ((ABS) SEG OPENCODE NIL ((COND ((MINUSP X) (- X)) (T X))))
 ((ABS) SEGTYPE SIMPLE)
 ((ABS) INPUT NIL X NIL)
 ((ABS) OUTPUT NIL RESULT NIL)
 (((ABS) INPUT NIL X NIL) IO FREE-VAR X NIL)
 (((ABS) OUTPUT NIL RESULT NIL) IO RETURNVAL NIL NIL)

 ((ABS) SUBSEG (MINUSP))
 ((ABS) SUBSEG (-))
 ((ABS) SUBSEG (JOIN))
 ((ABS) DFLOW ((ABS) INPUT NIL X NIL) ((MINUSP) INPUT NIL X NIL))
 ((ABS) DFLOW ((ABS) INPUT NIL X NIL) ((-) INPUT NIL X NIL))
 ((ABS) DFLOW ((-) OUTPUT NIL RESULT NIL) ((JOIN YES) INPUT ID X1 RESULT))
 ((ABS) DFLOW ((ABS) INPUT NIL X NIL) ((JOIN NO) INPUT ID X2 RESULT))
 ((ABS) DFLOW ((JOIN) OUTPUT NIL RESULT NIL) ((ABS) OUTPUT NIL RESULT NIL))
 ((ABS) CFLOW (MINUSP YES) (-))
 ((ABS) CFLOW (MINUSP NO) (JOIN NO))
 ((ABS) CFLOW (-) (JOIN YES)))
```

Figure 17: Plan facts corresponding to Figure 16.

A `SEGTYPE` fact specifies how a segment interacts with control flow. A `SIMPLE` segment receives control flow from one place and sends control flow to one place. A `SPLIT` segment makes a choice. It receives control flow from one place and sends it to one of two or more places. A split has two or more cases which correspond to the different control environments initiated by the segment. A `JOIN` segment receives control flow from one of two or more other places and sends it to one place. Like splits, joins have cases which correspond to the different control environments which preceded the segment. (The utility of joins will be discussed below.)

A `WHEN` fact specifies the circumstances under which the various cases of a split are activated. The fact contains a logical expression which refers to the inputs of the split. The case is activated when the expression is true.

As an example of the facts which describe the basic features of a segment consider the following facts describing the segment MINUSP in Figure 17. The segment is a function call which calls the standard Lisp function MINUSP. The segment is a split with two cases. The YES case is activated when the input X is less than zero. Otherwise, the NO case is activated.

```
((MINUSP) SEG FUNCALL MINUSP ((MINUSP X)))
((MINUSP) SEGTYPE SPLIT)
((MINUSP YES) WHEN (< X 0))
((MINUSP NO) WHEN (NOT (< X 0)))
```

The second class of facts in a surface plan consists of two different facts which identify the ports of a segment. INPUT and OUTPUT facts specify the input ports and output ports respectively of a segment. (On a given segment, all of the port names must be unique.)

In addition, port facts specify some additional information about the port. In particular they specify whether the port is identical to or a subpart of some other port on the same segment. If the port is related to another port then the ID field of the fact specifies the name of the other port.

As a simple example of port facts, consider the ports of the segment "-" in Figure 17. These are both simple ports which are unrelated to other ports. The use of relationships can be seen in the description of the segment JOIN and will be discussed below.

```
((-) INPUT NIL X NIL)
((-) OUTPUT NIL RESULT NIL)
```

The third class of facts in a surface plan consists of two different facts which describe additional features of ports. An IO fact specifies three pieces of information about a port. First, it says what kind of port it is. LAMBDA-ARG specifies that the port is an input argument to a function definition. ACTUAL-ARG specifies that the port is an input argument to a function call. FREE-VAR specifies that the port is implemented as a free variable. RETURNVAL specifies that the port is the return value of the segment. SIDE-EFFECT specifies that the port is not directly referenced by the segment, but is modified due to side-effects performed by the segment.

If a port is a LAMBDA-ARG then the IO fact specifies the name of the argument. If a port is a FREE-VAR then the IO fact specifies the name of the variable. If a port is an ACTUAL-ARG (or SIDE-EFFECT) then the IO fact specifies the name of the variable (if any) which carries the value to (or from) the port in the program code. If a port is a LAMBDA-ARG or an ACTUAL-ARG then the IO fact specifies the argument number corresponding to the port.

A CREATEVAR fact is used to indicate that the return value of a segment has been assigned to a variable by an assignment operation. The names of variables are recorded in port description facts in order to help the coder module choose useful variable names when creating code corresponding to a plan.

As a simple example of port description facts, consider the facts describing the ports of the segment "-" in Figure 17. X is the first argument to the function being called. RESULT is the return value. Note that these facts have the port facts above embedded in them. There is no example of a CREATEVAR fact in Figure 17.

```
(((-) INPUT NIL X NIL) IO ACTUAL-ARG X 1)
(((-) OUTPUT NIL RESULT NIL) IO RETURNVAL NIL NIL)
```

The fourth class of facts in a surface plan consists of four different facts which describe the interrelationships between segments in a plan. A SUBSEG fact specifies that one segment is contained in another. For example, the segment MINUSP is inside the segment ABS.

A RECURSIVE fact specifies that a segment is a recursive instance of a containing segment. For example the plan for a recursive implementation of factorial would have an inner segment corresponding to the recursive call on the function. There is no example of a recursive fact in Figure 17.

A DFLOW fact specifies data flow from one port to another. It points directly to the source and destination ports.

A CFLOW fact specifies the control flow connecting segments in a plan. Note that control flow is only used where necessary in order to specify the effects of splits and joins. In other situations the ordering constraints specified by data flow links are sufficient.

As an example of data flow and control flow, consider the links (reproduced below) impinging on the join in Figure 17. First of all notice that the inputs of the join are located in different cases of the join. This reflects the fact that in a given execution of the program only one of the inputs will be supplied. ID relationships between the ports specify that the input values are propagated directly to the output of the join.

```
((JOIN) SEG JOIN NIL NIL)
((JOIN) SEGTYPE JOIN)
((JOIN YES) INPUT ID X1 RESULT)
((JOIN NO) INPUT ID X2 RESULT)
((JOIN) OUTPUT NIL RESULT NIL)

((ABS) CFLOW (MINUSP NO) (JOIN NO))
((ABS) CFLOW (-) (JOIN YES))

((ABS) DFLOW ((-) OUTPUT NIL RESULT NIL) ((JOIN YES) INPUT ID X1 RESULT))
((ABS) DFLOW ((ABS) INPUT NIL X NIL) ((JOIN NO) INPUT ID X2 RESULT))
((ABS) DFLOW ((JOIN) OUTPUT NIL RESULT NIL) ((ABS) OUTPUT NIL RESULT NIL))
```

The control flow facts impinging on the join specify that when control flow comes from the segment "-", the YES case of the join is activated, and when control flow comes from the NO case of the split MINUSP, then the NO case of the join is activated. The data flow specifies where the data flow comes from in these two situations and that the output of the join becomes the output of the segment ABS.

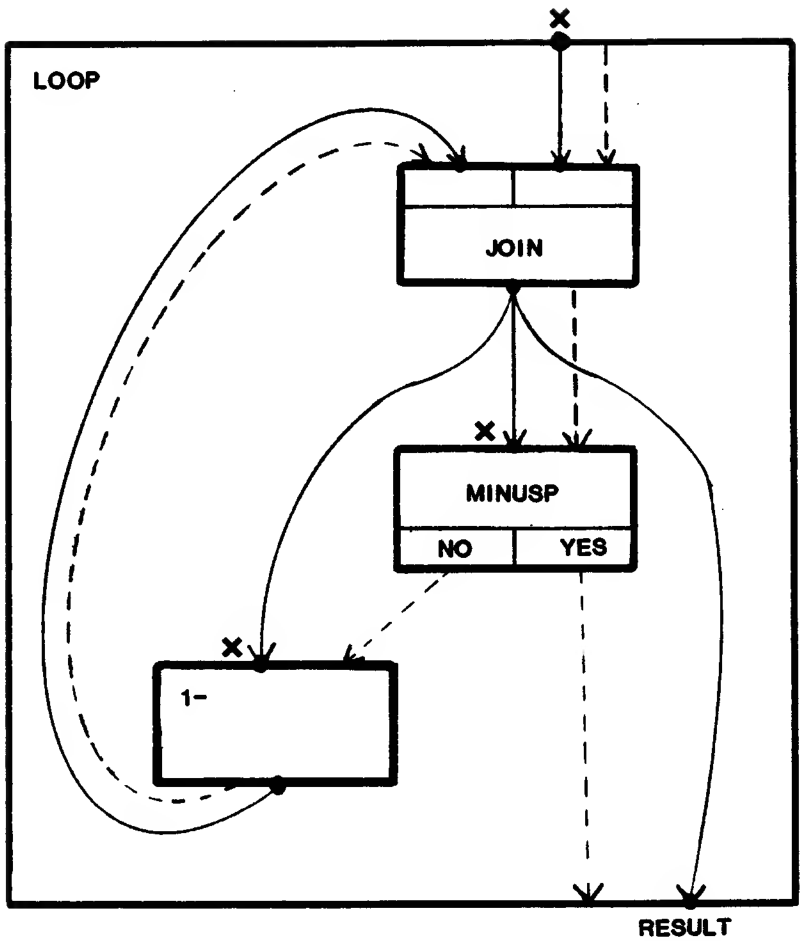

Figure 18: Looping plan for (LOOP DO (IF (MINUSP X) (RETURN)) (SETQ X (1- X))).

An interesting issue involving surface plans is the way loops are rendered. As shown in Figure 18 and Figure 19, this can be done in two ways. Both of these figures show plans for the trivial loop (LOOP DO (IF (MINUSP X) (RETURN)) (SETQ X (1- X))). The two figures differ in that the first represents the loop by using a loop in control flow and the second represents the loop by means of recursion.

The recursive plan is an abbreviated form of an infinite representation. The recursive link (represented by using a looping line) between the segments BODY and REC indicates that the computation performed by the segment REC can be described by exactly the same plan which describes the segment BODY. This corresponds to a recursive call of a procedure. The intermediate segment BODY is introduced (rather than making REC be a recursive instance of the segment LOOP) in order to provide a control environment where loop initializations can be placed (i.e., inside the segment LOOP, but before the segment BODY).

The fact that both figures represent the same computation reflects the fact that any loop can be represented as a tail recursion and vice versa. As will be seen below, KBEmacs usually chooses to represent loops in terms of recursion. However, this is not a requirement of the plan formalism *per se*.

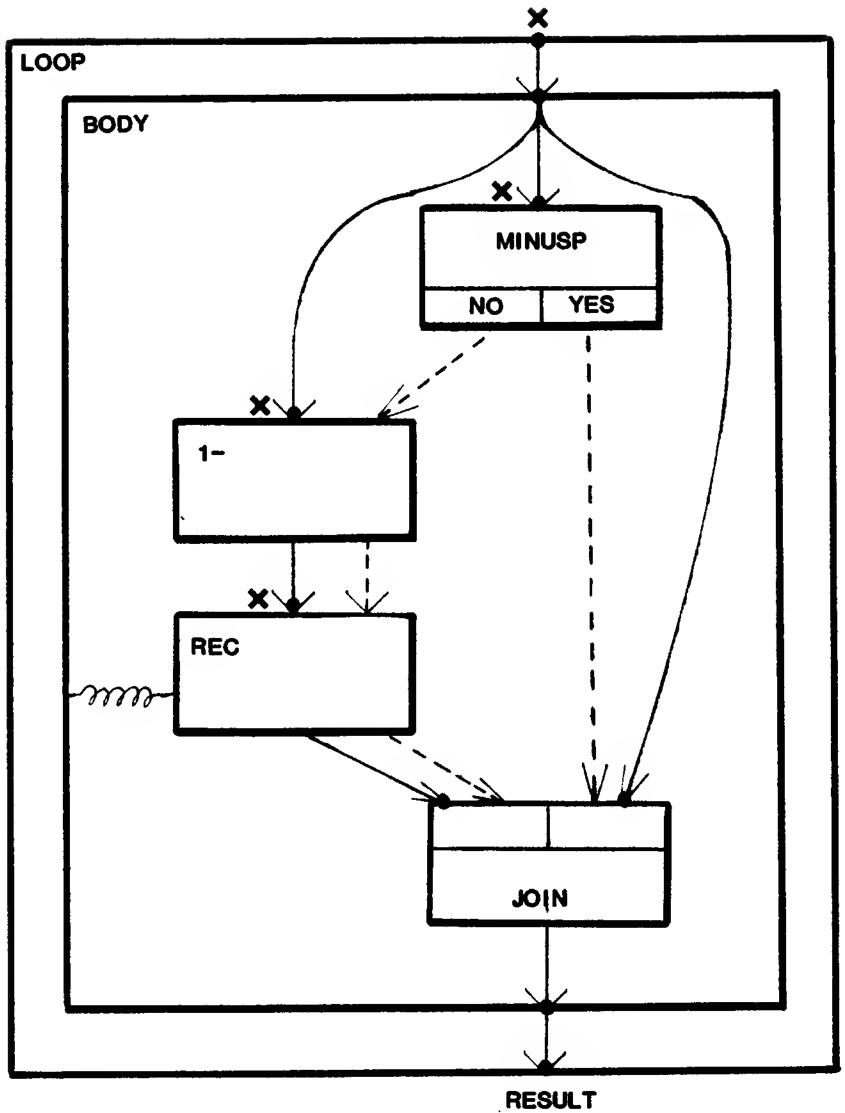

Figure 19: Recursive plan for `(LOOP DO (IF (MINUSP X) (RETURN)) (SETQ X (1- X)))`.

A final issue involving surface plans is their expressiveness. In a surface plan one can already see most of the roots of the limits to the expressiveness of the plan formalism. In particular, there are no facts which say anything about data structures or specifications. There is also no way to describe non-local flow of control such as interrupts or the Lisp operations `CATCH` and `THROW`. It is however, possible to represent multiple entry loops and multiple and mutual recursion. The ability to represent such complex patterns of control is lost when additional features are added into the plan formalism.

Grouped Plans

Typically, surface plans are flat in that they have no hierarchical structure. Rather, they are composed of an outermost segment containing tens (or even hundreds) of terminal subsegments connected by a network of data flow and control flow.

In a grouped plan, intermediate segments are used in order to break the plan up into a hierarchy of plans within plans. The overall goal of this grouping is to divide the plan into chunks which can, to a considerable extent, be manipulated separately. To this end the grouping is guided by the following principles. Segments

which are closely related (by control flow or data flow) are grouped together. Perhaps even more importantly, unrelated segments are kept as far apart as possible.

When experimenting with various ways to group up a plan, it was discovered that there are only a very few basic hierarchical organizations which are used in programs. These organizations are referred to in the plan formalism as *plan building methods* [Waters 78,79]. Each plan building method corresponds to a stereotyped way of combining subsegments together. The individual subsegments are identified by the roles they play in the combination.

Plan building methods are an important underpinning of KBEmacs. In particular, both the analyzer and coder modules contain a significant amount of procedurally embedded knowledge about how to deal with each different plan building method. Further these modules are not capable of dealing with plans which do not fall into these categories.

The basic grouping of a plan (i.e., the hierarchy of segments within segments) is represented using the SUBSEG fact discussed in the last subsection. Figure 20 shows the facts which are used to specify information about the plan building methods for the segments in a plan.

```
fact ::= ... | pbm-fact
pbm-fact ::= (segcase PLANTYPE plantype)
           | (segcase PBM-ROLE roleseg pbm-role)
plantype ::= JOIN | CONSTANT | BUILTIN | EXPR | AND | XOR | PRED | SSR | SSR-BODY
roleseg ::= segcase
pbm-role ::= ACTION | PRED | JOIN | INIT | REC | OP | BODY
```

Figure 20: The information in a grouped plan.

A PLANTYPE fact specifies the plan building method corresponding to a segment. A PBM-ROLE fact specifies the role each subsegment plays in the plan building method for the segment which contains it. As shown below, each plan building method has a particular set of roles associated with it.

plan building method	roles
JOIN, CONSTANT, & BUILTIN	*no subsegments*
EXPR & AND	ACTIONs
XOR	PRED, ACTIONs, & JOIN
PRED	INIT, PREDs, & JOINs
SSR	INIT, PRED, OP, REC, JOIN, & BODY (an SSR-BODY)

JOIN, CONSTANT, and BUILTIN are degenerate plan building methods for terminal segments. They correspond to joins, constants and function calls respectively. They are included so that every segment will have a plan building method.

The plan building method EXPR contains an arbitrary number of subsegments referred to as ACTIONs. An EXPR cannot contain any splits or joins, and the actions must be connected by an acyclic net of data flow. The plan building method EXPR corresponds to what is commonly thought of as an expression in a programming language.

The plan building method AND (which has nothing to do with the Lisp function AND) is a special case of the plan building method EXPR. It has the further requirement that there most be no data flow between the actions. It embodies the key idea that there are no ordering constraints between the actions.

The plan building method XOR corresponds to a conditional. It has a predicate which decides which of several actions to perform. A JOIN combines together the control flow which is split apart by the predicate. The segment ABS in Figure 16 is an example of an XOR. Figure 21 shows the facts which would be used to encode the plan building method information associated with Figure 16.

```
(((MINUSP) PLANTYPE BUILTIN)
 ((-) PLANTYPE BUILTIN)
 ((JOIN) PLANTYPE JOIN)
 ((ABS) PLANTYPE XOR)
 ((ABS) PBM-ROLE (MINUSP) PRED)
 ((ABS) PBM-ROLE (-) ACTION)
 ((ABS) PBM-ROLE (JOIN) JOIN))
```

Figure 21: Grouping facts corresponding to Figure 16.

The plan building method PRED corresponds to a compound predicate. It combines simple predicates with joins and initialization computation in order to create more complex predicates.

The plan building method SSR (Single Self Recursion) corresponds to a single self recursive program such as the standard implementation of factorial. The plan building method SSR is also used to represent loops which (as discussed above) are expressible as tail recursive computations. Figure 19 is an example of an SSR. Figure 22 shows the facts which would be used to encode the plan building method information associated with Figure 19.

```
(((MINUSP) PLANTYPE BUILTIN)
 ((1-) PLANTYPE BUILTIN)
 ((JOIN) PLANTYPE JOIN)
 ((REC) PLANTYPE SSR-BODY)
 ((BODY) PLANTYPE SSR-BODY)
 ((LOOP) PLANTYPE SSR)
 ((LOOP) PBM-ROLE (MINUSP) PRED)
 ((LOOP) PBM-ROLE (1-) OP)
 ((LOOP) PBM-ROLE (REC) REC)
 ((LOOP) PBM-ROLE (JOIN) JOIN)
 ((LOOP) PBM-ROLE (BODY) BODY))
```

Figure 22: Grouping facts corresponding to Figure 19.

An SSR has several different kinds of roles. In particular, it has a BODY (which has the special plan building method SSR-BODY) and a REC (which is a recursive instance of the BODY and therefore also has the plan building method SSR-BODY). It can have an INIT which performs initialization computation before the BODY is entered. It typically has an OP which is the basic computation performed by the SSR. If an SSR is capable of terminating then it will have a PRED which specifies when the termination will occur.

A wide range of programs can be analyzed in terms of the nine plan building methods above. However, there are other ways in which a program can be organized. This causes grouped plans to be less expressive than surface plans. In order to recover the full expressiveness of surface plans, plan building methods corresponding to these other organizations would have to be added into the plan formalism. (Unfortunately, this is relatively hard to do because it requires a considerable amount of procedural support in the analyzer and coder.)

As examples of program organizations not covered by the plan building methods presented above consider the following. There are no plan building methods which correspond directly to loops like the one in Figure 18. In general, this is not a problem because most loops can be transformed to equivalent recursive programs as shown in Figure 19. However, this transformation cannot be applied to multiple entry loops. In addition, occasionally the control flow in a program is so complex that it is not possible to sensibly analyze it in terms of conditionals and loops at all. In this situation it would often be useful if there were a plan building method which could view the program as a finite state automaton with state transitions implemented as control flow. Finally, the plan formalism does not have plan building methods corresponding to multiple or mutual recursion. (However, for many purposes, these can be satisfactorily modeled in the plan formalism by

merely representing the recursive function calls in the same way as non-recursive function calls.)

The discussion of plan building methods above is intentionally brief. The discussion in [Waters 78] is much more detailed. In addition, the latter discussion shows a number of examples of hierarchical plans which are significantly more complicated than the ones above.

In closing, it should be noted that, at a logical level, plan building methods are clearly cliches. Each one specifies how a set of roles is to be embedded in a matrix connecting them. Unfortunately, the restrictions embodied in a plan building method are not specific enough for them to be represented as a cliche in the plan formalism. There are two main problems. First, the number of roles is not fixed — an EXPR can have arbitrarily many ACTIONs. Second, the matrix is not fixed, rather it is only constrained — an EXPR can have any pattern of data flow as long as it is acyclic.

It is unfortunate that plan building methods cannot simply be represented as cliches in the plan formalism, because this forces them to be procedurally handled as a special case. It should be noted that the plan calculus is also incapable of representing plan building methods as cliches because, in general, it too requires the number of roles and the matrix to be fixed. However, this may not be a problem because the next demonstration system may not need to use plan building methods. The possibility of eliminating the plan building methods will be discussed in the last section of this chapter.

Temporal Plans

As discussed in Chapter I, a particularly interesting aspect of the plan formalism is the way it represents loops. As shown above, loops can be represented either using loops in control flow or recursively. In addition, a special mechanism called *temporal decomposition* is provided which makes it possible to represent a loop as a composition of *temporal fragments* which communicate by means of *series* of values. A series of values is a sequence (stream) of values generated over a period of time. Temporal fragments produce and consume series of values.

As an example of temporal fragments, Figure 23 shows a plan diagram for a program SUM-TO-N which adds up the first N integers. This program (shown below) computes the sum by counting down from N to zero and adding up the integers enumerated.

```
(DEFUN SUM-TO-N (N)
  (LET ((I N)
        (RESULT 0))
    (LOOP DO
      (IF (MINUSP I) (RETURN RESULT))
      (SETQ RESULT (+ RESULT I))
      (SETQ I (1- I)))))
```

The plan in Figure 23 contains two temporal fragments ENUMERATE-FROM-N and SUM. Each of these fragments is a simple recursive plan. Note that the plan for the segment ENUMERATE-FROM-N is identical to the plan for the segment LOOP in Figure 19 except that it does not return a value.

The series which are produced and consumed by a temporal fragment are represented by a special kind of input/output port called a *temporal* port. The ports I and J in Figure 23 are temporal ports. The temporal output I corresponds to the series of integers that is created by counting down from N. The dashed line between the NO case of the segment MINUSP and the port I indicates that the temporal port only corresponds to the series of integers which are greater than or equal to zero. The temporal input J corresponds to the series of values which is summed up.

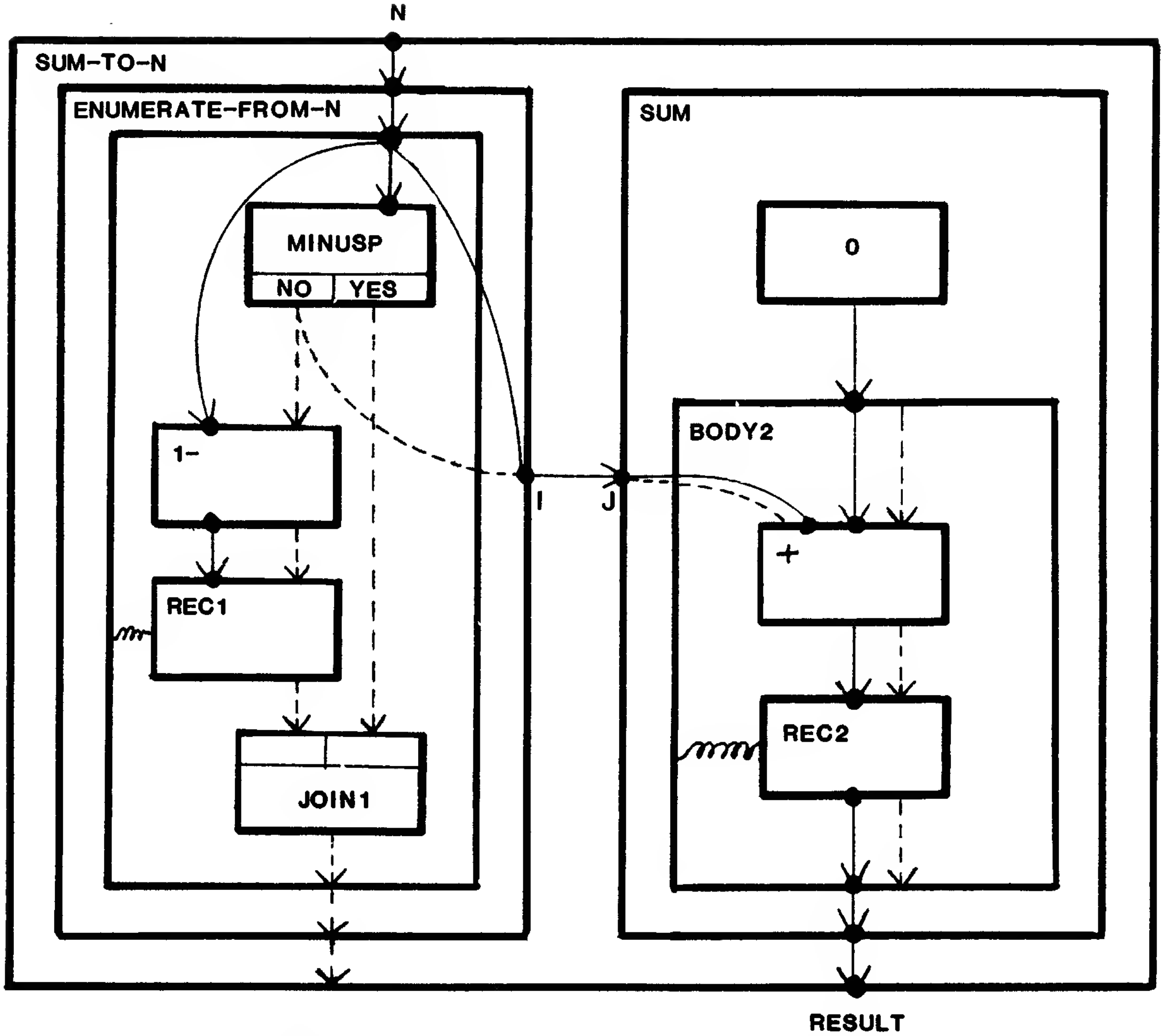

Figure 23: Temporal plan for summing the integers from `1` to `N`.

Figure 24 presents the plan facts which are used to represent temporal fragments. First, two new kinds of ports are defined — temporal inputs (`TINPUT` facts) and temporal outputs (`TOUTPUT` facts). These facts have an object name field like any other port. In addition they have two other fields which describe the series of values corresponding to the port. The ***lower-io*** field specifies a port inside of the temporal fragment. The series of values corresponding to the temporal port is the series of values over time of the ***lower-io*** port. (In a plan diagram a temporal port is linked to its ***lower-io*** port by a solid line with no arrowhead.)

The ***lower-env*** field specifies a segcase inside of the temporal fragment. Usually it is the segcase of the ***lower-io*** port. However, when it is not, it specifies that the series of values is only the subset of values of the ***lower-io*** which is visible in the ***lower-env*** control environment. (In a plan diagram a temporal port is linked to its ***lower-env*** port by a dashed line with no arrowhead.)

Figure 24 also shows that a new kind of plan building method `TEMPCOMP` (temporal composition) is provided. This plan building method is used for segments which contain temporal fragments. (The temporal fragments themselves are instances of the plan building method `SSR`.) A `TEMPCOMP` is essentially the same as an `EXPR` except that it contains temporal fragments.

Several new plan building method roles are defined in order to specify the roles of temporal fragments in a `TEMPCOMP`. An `ENUM` enumerates a series of values, terminating the loop when the series is exhausted. A `GEN` generates a series of values, but does not contain any termination condition. An `ACC` takes in a series of values

and accumulates some result. A TRN is a transducer which takes in a series of values and creates a new series of values based on the input series. A TERM contains a termination condition which can cause early termination of a loop. It can be used to create multiple exit loops. A FILTER takes in a series of values and filters out some of these values, creating a new series which contains only some of the values in the input series. Finally the role BSSR is used for any kind of complex fragment which is too complex to fit in one of the categories above.

```
port ::= ... | (segcase TINPUT lower-env obj lower-io)
             | (segcase TOUTPUT lower-env obj lower-io)
lower-env ::= segcase
lower-io ::= port
plantype ::= ... | TEMPCOMP
pbm-role ::= ... | ENUM | GEN | ACC | TRN | TERM | FILTER | BSSR
io-kind ::= ... | DUMMY
```

Figure 24: The information in a temporal plan.

The facts IO and CREATEVAR are used to describe features of temporal ports just as they are used to describe ordinary ports. In addition, Figure 24 shows that there is a special kind of port descriptor (DUMMY) which only applies to temporal ports. This is used for temporal ports which have to be present in order to correctly represent the interrelationships between temporal fragments but for which no data is actually required. These correspond exactly to the places where the notation {..., depending on ...} has to be used (e.g.. in the cliche count discussed above).

As an example of facts describing the interaction of temporal fragments, consider the facts in Figure 25. This figure shows the facts which deal with the temporal aspects of the plan in Figure 23. The segment SUM-TO-N is a temporal composition in which ENUMERATE-FROM-N is an enumerator and SUM is an accumulator. I and J are temporal ports as described above. Note that ordinary data flow is used to connect temporal ports.

```
(((SUM-TO-N) PLANTYPE TEMPCOMP)
 ((SUM-TO-N) PBM-ROLE (ENUMERATE-FROM-N) ENUM)
 ((SUM-TO-N) PBM-ROLE (SUM) ACC)
 ((ENUMERATE-FROM-N) TOUTPUT (BODY1) I ((BODY1) INPUT NIL I NIL))
 ((SUM) TINPUT (+) J ((+) INPUT NIL X NIL))
 (((ENUMERATE-FROM-N) TOUTPUT (BODY1) I ((BODY1) INPUT NIL I NIL))
  IO FREE-VAR I NIL)
 (((SUM) TINPUT (+) J ((+) INPUT NIL X NIL)) IO FREE-VAR I NIL)
 ((SUM-TO-N) DFLOW
            ((ENUMERATE-FROM-N) TOUTPUT (BODY1) I ((BODY1) INPUT NIL I NIL))
            ((SUM) TINPUT (+) J ((+) INPUT NIL X NIL)))
```

Figure 25: The temporal facts corresponding to Figure 23.

The discussion above gives only a brief overview of temporal fragments. They are discussed in much greater detail in [Waters 78,79]. The section on the analyzer module below discusses how KBEmacs decides to break a loop up into temporal fragments.

Cliches

There are three kinds of facts in the plan formalism which are used to represent information about roles and cliches (see Figure 26). A CLICHE-ROLE fact is used to specify that a segment is a role. It specifies the name and the type of the role. A CLICHE-DEF fact specifies that a segment is the definition of a cliche. In addition, it records the declaration information associated with the definition. A CLICHE fact specifies that a segment is an instance of a cliche. (Note that the non-terminal symbol ***cliche-declaration*** is defined in Figure 12.)

```
fact ::= ... | cliche-fact
cliche-fact ::= (segcase CLICHE-ROLE roleseg name cliche-role-type)
              | (segcase CLICHE-DEF name ({cliche-declaration}))
              | (segcase CLICHE name)
cliche-role-type ::= INPUT | OUTPUT | INTERNAL
```

Figure 26: The information in a cliche plan.

As examples of the first two kinds of facts above, consider the cliche absolute-value shown below.

```
(DEFINE-CLICHE ABSOLUTE-VALUE
    (PRIMARY-ROLES NUMBER
     DESCRIBED-ROLES NUMBER
     COMMENT "computes the absolute value of {number}")
  (LET ((X {input number}))
    (COND ((MINUSP X) (- x)) (T X))))
```

Figure 29 shows a plan diagram for this cliche. Figure 27 shows the cliche definition facts corresponding to the cliche.

```
(((ABSOLUTE-VALUE) CLICHE-DEF ABSOLUTE-VALUE
                   (PRIMARY-ROLES NUMBER
                    DESCRIBED-ROLES NUMBER
                    COMMENT "computes the absolute value of {number}"))
 ((ABSOLUTE-VALUE) CLICHE-ROLE (NUMBER) NUMBER INPUT))
```

Figure 27: Cliche definition facts corresponding to Figure 29.

As an example of facts describing the instance of a cliche consider the following. Suppose that the segment ABS in Figure 29 were an instance of the cliche SIMPLE-CONDITIONAL shown below.

```
(DEFINE-CLICHE SIMPLE-CONDITIONAL
    (PRIMARY-ROLES TEST ACTION
     DESCRIBED-ROLES TEST ACTION
     COMMENT "computes {action} if {test} is true")
  (IF {test} {action}))
```

If this were the case then the plan for the segment ABS would contain the facts shown in Figure 28. Note that the fact CLICHE-ROLE is used to represent both roles in cliche definitions and roles in cliche instances.

```
(((ABS) CLICHE SIMPLE-CONDITIONAL)
 ((ABS) CLICHE-ROLE (MINUSP) TEST INTERNAL)
 ((ABS) CLICHE-ROLE (-) ACTION INTERNAL))
```

Figure 28: Cliche instance facts corresponding to Figure 29.

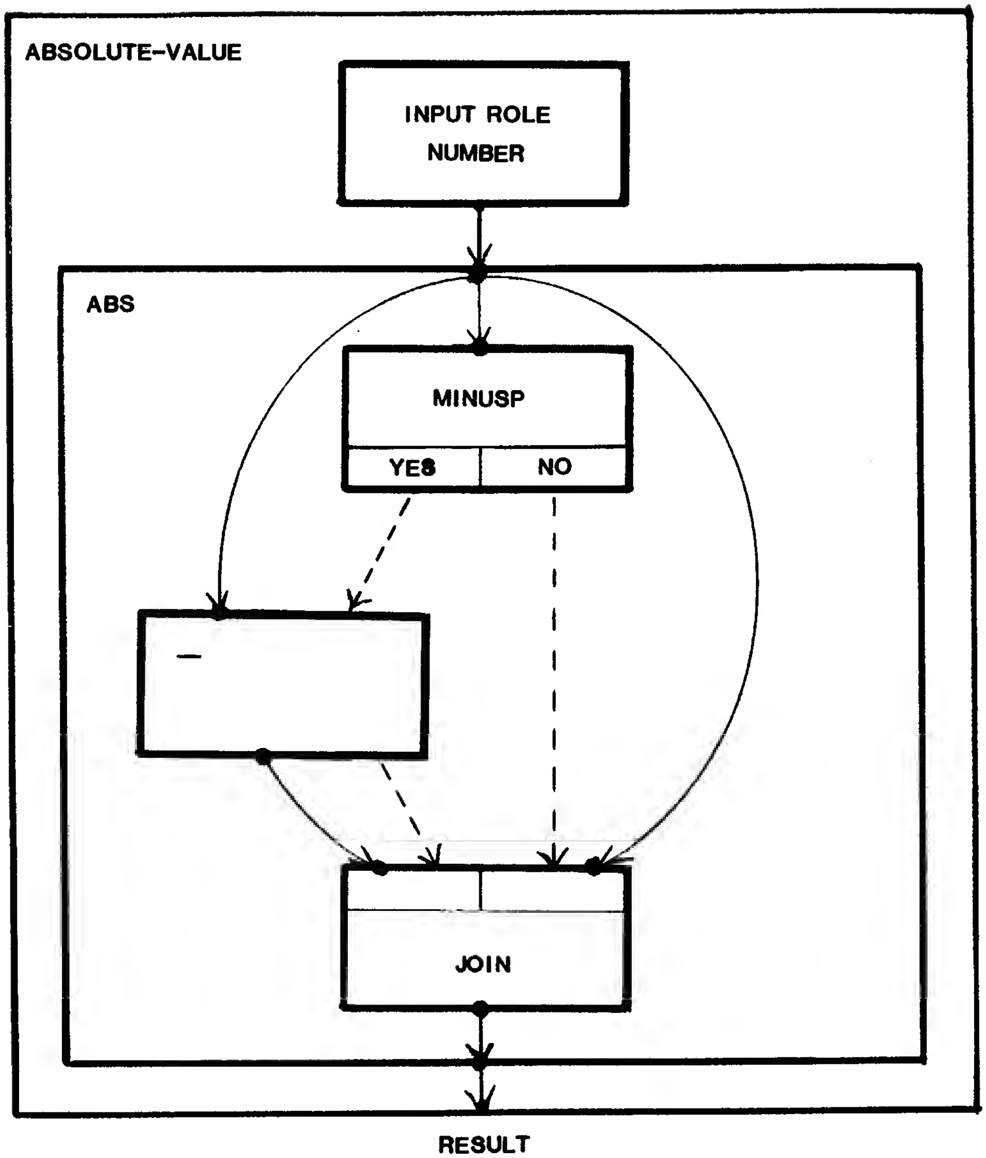

Figure 29: A plan for the cliche absolute-value.

As can be seen from the discussion above, it is in general very straightforward to represent information about cliches in the plan formalism. However, there is one area of complexity — compound roles. Compound roles are represented by `CLICHE-ROLE` facts just like non-compound roles. For example suppose that a top level segment `T` contains a segment `S` which is the role `{the step of the enumerator}`, and that the intermediate segment `E` is `{the enumerator}`. In this case, two `CLICHE-ROLE` facts are used one specifying that `E` is the enumerator of `T` and one specifying that `S` is the step of `E`.

In order for the approach above to work, the plan formalism has to require that there always be a segment which can be identified to correspond to a given compound role. This requirement is problematical due to the fact that it cannot always be satisfied. Some sets of roles are logically grouped inside segments and some are not. (This is not a severe problem in KBEmacs at the current time because there are actually very few cases of compound roles.)

Requiring that there be a segment corresponding to each compound role is not merely a convenience — it solves an important problem. In addition to knowing what roles are inside of a compound role, KBEmacs needs to know what part of the matrix of the cliche is inside of the compound role. However, there is not any

kind of annotation which directly specifies this information. As a heuristic, KBEmacs assumes that all of the matrix inside of the lowest level segment which contains all of the sub-roles in the compound role is also inside of the compound role. The importance of this will be discussed further below when the issue of filling roles is discussed.

Implementation

The plan formalism is supported by a simple relational data base. Individual facts can be asserted, deleted, and retrieved with patterns. For example, the following form asserts a fact describing an input.

```
(>& '(ABS) INPUT NIL 'X NIL)
```

The next form retrieves a list of all the input facts for the segment ABS. The symbol * is used as a wild card matching symbol. In order to access the fields of the facts which are retrieved by a pattern, various field accessing functions are used.

```
(??& '(ABS) INPUT * * *)
```

Finally, the next form finds and deletes any and all input facts describing the input object X of the segment ABS.

```
(--& '(ABS) INPUT * 'X *)
```

The above data base is very flexible. For example, the use of patterns allows for great variety in the way information can be retrieved. In addition, it is trivial to define new kinds of facts. This flexibility is an important virtue in as much as the focus of the KBEmacs implementation has been rapid prototyping and experimentation.

Unfortunately, the flexibility in the implementation of the plan data base is bought at the cost of horrendous inefficiency in both time and space. The last section of this chapter discusses how reimplementing the data base in a much more rigid way could speed up information access by orders of magnitude.

Language Independence

An important virtue of the plan formalism is that it is inherently programming language independent. This provides the basis for the language independence of KBEmacs as a whole.

Analyzer

The analyzer is used in two situations. First, whenever the programmer textually modifies the program being worked on, the analyzer creates a new plan for the program so that knowledge-based editing can continue. Second, the analyzer is used to process cliche definitions in order to create plans which are stored in the cliche library. An important feature of the analyzer in both situations is the ability to understand role annotation.

The analyzer operates in a sequence of stages which are described below. Many of these stages are similar to the operations of an optimizing compiler, and will only be sketched out briefly. [Waters 78] provides much more detail.

Parsing

The first stage of the analyzer parses the program text to be analyzed. For Lisp, this parsing is performed by the Lisp reader. The only direct support which has to be supplied by the analyzer is the parsing of role annotation. This is done by defining a special reader macro for the character "{". For other languages, a parser has to be implemented along traditional lines.

Semantic Processing

The second stage of the analyzer translates the parse tree created by the first stage into an equivalent program in a simple intermediate language. The purpose of this translation is to convert all complex control constructs and data constructs in the the source language into simple operations. All of the control constructs are converted to simple conditional and unconditional branches. All of the data constructs are converted into binding and assignments of simple atomic variables.

The translation is done through a process of macro expansion. This process expands any macro calls in the source program. In addition, it uses a set of additional macro-like definitions which expand each complex construct in the source into simpler forms. The set of macro-like definitions for the complex forms act as a semantic definition for these constructs.

Creating a Surface Plan

The third stage of the analyzer takes the intermediate representation created by the semantic stage and creates a surface plan corresponding to it. The surface plan stage is implemented as a symbolic evaluator which follows every control path in the program, creating segments, data flow, and control flow as it goes.

Each time a function call or role is encountered, an appropriate segment is created. In addition, data flow is created which links this segment with the appropriate arguments, if any. Each time a conditional branch is encountered, a split segment is created, and the symbolic evaluator proceeds along both paths indicated by the conditional branch. Each time a branch steers the symbolic evaluator onto a path which has already been processed, then a join is created and the symbolic evaluator is prevented from reprocessing the path.

Grouping

The fourth stage of the analyzer takes the surface plan generated by the third stage and creates a grouped plan by inserting appropriate intermediate segments. As the first step of this process, the grouping stage locates each loop in the surface plan and converts it into an equivalent recursive representation. As part of this, the grouping stage has to determine which data flow arcs are associated with the loop and therefore must be represented as arguments in the recursive representation. The existence of explicit join segments greatly facilitates this task.

Once the loops have been located, the grouping stage creates a grouped plan by parsing the surface plan in terms of the plan building methods. Each time a group of segments is located in the surface plan which interact in accordance with one of the plan building methods, these segments are gathered together into a single intermediate segment. As part of this, appropriate PLANTYPE and PBM-ROLE facts are created.

A priori, one might think that parsing in terms of plan building methods would be a difficult task requiring significant backtracking. However, it turns out that this is not the case. In fact, parsing is relatively straightforward and no backtracking is ever required. The key to this is the fact that there are only a very small number of plan building methods and they are so different from each other that it is very difficult to make a mistake about which one is applicable where.

The parsing process is driven primarily by the control flow in the surface plan. After the parsing is completed, the grouped plan is adjusted based on the data flow in the plan. In particular, each segment which creates a value is moved as close as possible (within the constraints of control flow) to the place where this value is used. This adjustment is done in order to further the goal of keeping closely related segments close together and unrelated segments far apart.

Temporal Decomposition

The fifth stage of the analyzer performs temporal decomposition. Each loop in the grouped plan is broken apart into temporal fragments. As discussed in [Waters 79], this decomposition is done based on the data flow

in the loop. The main goal is to locate fragments of the loop which can be understood in isolation. The key requirement is that all feedback of information from the fragment to the fragment must be contained within the fragment.

A significant weakness of the the temporal decomposition stage is that it only works for tail recursive programs (i.e., loops). The discussion in [Waters 78] indicates how temporal decomposition could be generalized to singly recursive programs which are not tail recursive. Generalizing temporal decomposition to multiply recursive programs is a topic of current research.

Locating Compound Roles

The final stage of analysis identifies which intermediate segments correspond to compound roles. This cannot be done until after the other stages of analysis have identified what grouping is appropriate.

Possible Improvements

The analyzer is the oldest and most robust module of KBEmacs. It is well tested and relatively bug free. Nevertheless, there are many ways in which it could be improved.

To start with, there are many limitations on the kinds of programs which the analyzer can analyze. Most of these limitations stem from limitations of the plan formalism (e.g., the fact that the plan formalism cannot represent non-local control flow and the fact that grouped plans cannot represent multiple entry loops). Other limitations stem from weaknesses of the analyzer itself. For example, the fact that temporal decomposition can only be applied to loops.

Finally, little attempt has been made to make the analyzer efficient. It should be possible to speed it up considerably by using some of the efficient graph algorithms which are employed by optimizing compilers.

Recognition

From the point of view of the PA project as a whole, the most glaring defect of the analyzer module is the fact that it is not able to recognize which cliches could have been used to construct a program.

When a program is constructed by means of knowledge-based editing using cliches, then the plan contains a record of the cliches which were used. Several of the capabilities of KBEmacs depend on the presence of this information (e.g., the "`Replace`" and "`Comment`" commands). Unfortunately, since analysis is not able to recover this information, these commands cannot be used after a program has been modified textually.

There is one situation where KBEmacs is able to avoid the problem above. If KBEmacs is able to recognize that the only effect of a textual modification is to fill in a role, then KBEmacs converts the textual modification into an equivalent "`Fill`" command and is able to maintain its knowledge of how the program was built up out of cliches.

In principle, one would expect that it should be possible to parse a plan in order to determine the cliches which could have been used to create it just as a plan can be parsed in order to determine which plan building methods could have been used to create it. Brotsky [Brotsky 84] has developed an efficient graph parsing algorithm which should make this possible. Work (by Linda Zelinka) is currently under way on an analysis module for the next demonstration system which will support cliche recognition. This module is based on the work of Brotsky and will operate in terms of the plan calculus.

The primary reason that plan building methods were included as part of KBEmacs was to make it possible to create a grouped plan even though cliche recognition is not supported. This was done because it was assumed that having a grouped plan was essential for performing knowledge-based editing. However, experience has suggested that this is probably not the case. As will be discussed in the last section of this chapter, it would probably be possible to do without the concept of plan building methods if KBEmacs were reimplemented from scratch. In any event, the next demonstration system will certainly not need to use plan

building methods to create grouping since it will support cliche recognition.

Ada

The analyzer module supports the language Ada in addition to Lisp. However, its support for Ada is much less extensive than the support for Lisp. Basically, it supports all of those features of Ada which are semantically equivalent to Lisp, and not very much else.

In particular, data declarations are treated in a very minimal way. They are essentially processed as literals. The analyzer does nothing with them except save them. Note however, that even this very minimal support allows KBEmacs to have cliches for data declarations by simply having cliches which contain the appropriate declaration literals.

The only parts of KBEmacs which can be said to understand Ada declarations in any way are special purpose procedures in the coder module and in constraints. In order to provide better support for declarations, the plan formalism would have to be extended so that it could represent information about data structures.

A good example of the limits to KBEmacs understanding of declarations is demonstrated by its support for generic packages. On one hand, it understands almost nothing about them. On the other hand, it succeeds in making use of them in the scenario in Chapter III. KBEmacs contains an Ada interpreter which correctly interprets generic packages. In addition, there are several generic packages (shown in Appendix B) which are assumed to have been written before the scenario begins. All of this is basically outside of the understanding of KBEmacs. Within the understanding of KBEmacs there are several cliches which refer to generic packages (e.g., `CHAIN_FILE_DEFINITION` and `FILE_SUMMARIZATION`). These cliches make it possible for KBEmacs to create programs which use generic packages.

Another problematical aspect of the analyzer's support for Ada is the support for exception handlers. In Lisp, the analyzer does not handle interrupt processing code at all, since it cannot be represented in the plan formalism. In Lisp, this weakness is easy to avoid since relatively few Lisp programs use interrupts. However, interrupts cannot be so easily ignored in Ada programs. The analyzer provides minimal support for Ada exception handlers by converting them into special conditionals which are semantically incorrect, but representable in the plan formalism. A reverse conversion is then performed by the coder module.

A final difficulty with Ada is inherent in the language itself. Ada is extraordinarily difficult to parse. The biggest problem stems from difficulties in discriminating between various syntactic features (for example, array references are indistinguishable from function calls, zero argument function calls are indistinguishable from variable references, and qualified names are indistinguishable from record field references). Due to these and other confusions, it is impossible to fully parse a program unless a package declaration has been defined for every package which the program refers to. (Roles present a special problem since they are, of course, not defined in any package.) In addition to being a problem for KBEmacs, this parsing problem is a problem for the incremental construction of Ada programs in general.

The analyzer's support for Ada should really only be looked at as illustrative. There is however, no fundamental reason why more complete support could not be provided.

Language Independence

An important aspect of the analyzer is that it is largely programming language independent. Only the first two stages are language dependent. In order to support a new language one has to provide a parser for it and a set of macros which define the semantics of the constructs in the language in terms of the intermediate language used by the analyzer. In addition to Lisp and Ada, this has been done, at least to some extent, for the languages Fortran, Cobol, and PL/I.

Coder

The coder module is used to create new program text whenever the plan for the program being edited is changed. Like the analyzer, the coder operates in a sequence of stages most of which are relatively language independent.

Temporal Composition

The first stage of the coder reverses the process of temporal decomposition. All of the fragments inside of a TEMPCOMP are combined together into a single fragment which is then rendered as a simple recursive plan. This process is relatively straightforward and is described in detail in [Waters 78].

Grouping

The second stage of the coding process analyzes a plan in order to determine the basic control flow structures in the plan — i.e., the conditionals and loops. The plan is grouped up hierarchically based on this analysis. This grouping is important in order to determine which control constructs should be used when coding the program. As part of this, any segment which produces a value is moved as close as possible to the place where this value is used. This is important in order to be able to maximize the extent to which nesting of expressions can be used to implement data flow.

This grouping process is discussed here because it is logically part of the coding process. However, this grouping is actually performed by the analyzer. Instead of grouping a plan when it is going to be coded, KBEmacs makes sure that plans are always appropriately grouped. Whenever the analyzer makes a plan for a program or for a cliche it groups it. Whenever the knowledge-based editor combines plans together it makes sure that the result is appropriately grouped. In addition, the temporal composition stage calls parts of the analyzer in order to group up the loops it creates.

Implementing Data Flow

The third stage of the coder is in many ways the most difficult and the most important. It analyzes the grouped plan and determines how the data flow in the plan should be implemented. The stage first identifies all of the places where nesting of expressions can be used. It then selects variables to use for the other data flow.

Selecting variable names turns out to be very important and quite difficult. It is important because the readability of a program depends to a very large extent on how well the variable names are chosen. It is difficult because a large number of competing requirements have to be balanced against each other.

The data flow stage of the coder proceeds in several steps. First, it divides the data flow in the plan into sections such that all of the data flow in each individual section can be implemented in the same way.

Second, the data flow stage identifies which sections can be implemented using nesting of expressions. A significant complication to this is that deciding to use nesting of expressions in a given place often introduces ordering constraints in the plan. These have to be recorded so that they will not be contradicted later.

Third, the data flow stage tries to discover a reasonable variable name to use for the sections which cannot be implemented using nesting. In order to do this it consults the variable name suggestions in IO and CREATEVAR facts in the plan. These suggestions are made part of the plan precisely because they are essential for this purpose. Note that these suggestions come initially either from a cliche or from something typed by the programmer.

Often, several conflicting variable name suggestions will exist for a given section. When this is the case, a variety of heuristics are applied in order to pick the best name to use. In particular, informative names such as UNIT and SUM are preferred over uninformative names such as DATA and RESULT. Another key criterion is that names must be chosen so that the data flow in neighboring data flow sections will not interfere — if the

variable UNIT is used for a given section then it cannot be used for any other section which overlaps the given section in execution order.

Often, there are no suggestions of what variable name to use for a section. When this is the case the data flow stage attempts to use the same name as a preceding or following section which appears to correspond to the same logical object. If all else fails, a variable name is generated based on the name of the function that creates the value being transmitted or based on a list of acceptable uninformative variable names such as X, Y, and Z.

At the current time, the conflicting requirements and heuristics above are supported by an ad hoc procedure which makes the necessary choices serially without backtracking. This is relatively fast. However, it does not create markedly good results. In particular, this approach is unfortunately quite sensitive to the exact order in which the data flow sections come up for consideration. It would probably be preferable to use some kind of constraint propagation approach to come to a simultaneous consensus on all of the choices which have to be made.

Basic Coding

Once decisions have been made about how to implement the data flow in a plan the fourth stage of the coder creates code for the program. This is done recursively one segment at a time. For each terminal segment, code representing the appropriate function call, constant, or role is generated. For each intermediate segment, code is created for the control construct which corresponds to the segment's plan building method (e.g., a conditional is created for an XOR). The decisions of the third stage are followed in order to create data flow.

The result of the basic coding stage is a more or less language independent parse tree for the program. It is language independent in that it only relies on a few basic constructs (such as variable binding, conditionals, simple loops, variable assignments, and function calls) which most languages support. In this parse tree, roles are expressed as a special kind of function call.

Final Transformations

The fifth stage of the coder takes the language independent parse tree and performs a number of language dependent transformations on it. The goal of these transformations is to increase the aesthetics of the program produced. For example, in Lisp, these transformations introduce uses of the the special forms IF, WHEN, LET*, and LOOP.

In Ada, much more extensive transformations are supported. In particular, transformations reverse the processing which the analyzer uses to encode Ada declarations and exception handlers. Going beyond this, a special procedure inspects the program and creates variable declarations for any variables which do not have explicitly specified declarations. Processing to ensure that each data file is opened and closed appropriately in a program is also performed in this stage. (It would probably be more logical if both of these things were done as part of knowledge-based editing.)

A significant complexity in the transformation stage is that it must be careful to maintain the links between the plan and the parse tree so that KBEmacs will know what parts of the plan correspond to what parts of the corresponding program — e.g., in order to support the "Highlight" command.

Pretty printing

The sixth and final stage of the coder creates program text corresponding to the transformed parse tree. This is done using a pretty printer [Waters 83b,84b]. The flexibility of this pretty printer makes it possible to handle a wide variety of aesthetic issues at this final stage. In particular, the pretty printer is specifically designed to make it easy to define how a parse tree is to be unparsed. This is done by defining a set of

formatting functions corresponding to the various kinds of nodes in the parse tree.

Possible Improvements

As can be seen in the scenarios above, the coder module produces reasonable results. It works for completed programs, partial programs containing unfilled roles, and cliches. However, experimentation has shown that the coder is creaking on the edge of collapse. Several problems are particularly severe.

The biggest problem stems from the defects of the data flow stage. This stage does a good job of creating correct programs. However, it has a tendency to produce programs which, though correct, are unreadable. The fundamental problem is that the coder does not really have any understanding of what makes data flow aesthetic. It would probably be a considerable improvement to reimplement this stage using some kind of constraint propagation as discussed above. However, it is not clear that that would be a complete solution to the problem.

Another problem is that the transformational stage is not supported in a general way. Each transformation has to be implemented as a procedure which operates directly on parse trees. It would be better to write a general transformational support system which made it easier to state individual transformations. In addition, there might have to be some kind of support for deciding which transformation to apply when several are applicable.

An interesting problem with the coder is that it is not able to make good use of macros in the Lisp code it produces. A few macros are introduced by transformations which are specifically designed to introduce them, however, there is no general mechanism for ensuring that user defined macros will be used. It is probably not possible to use macros effectively in the absence of a general recognition mechanism which can reliable detect where computation which corresponds to a macro exists in the plan for a program.

A final problem is that of *stability*. When a programmer makes a small change in a program he wants to see only a small change in the code. In particular, he does not want to see any changes at all in the parts of the code which are not effected by the change. This important property is only provided indirectly by the coder. In the examples tested so far, stability is achieved because the coder processes similar programs in similar ways. However, there is not any part of the coder which explicitly checks to see that stability is achieved. As a result, it is not clear that it really is achieved in any robust way.

A problem related to the above is that the programmer cannot, in general, control the style of the code produced. If the programmer makes some purely stylistic (as opposed to semantic) change in the program text, then the coder will simply restore the text to its old form the next time it codes the plan. The only exception to this is variable names which are the only place where the coder explicitly takes advice from the programmer.

Language Independence

An important aspect of the coder is that it is largely programming language independent. To a considerable extent, only the last two stages are language dependent. In order to support a new language, one has to define an appropriate set of transformations which introduce language specific forms into the language independent parse tree created by the first four stages of the coder. In addition, one has to provide an unparser for the language. This latter task is facilitated by the existence of the powerful pretty printer discussed above.

In addition to Lisp, the coder supports all of the aspects of Ada which the analyzer supports. However, if the analyzer were extended then the coder would have to be extended too. In addition, the coder provides basic support for the language PL/I.

Knowledge-Based Editor

The knowledge-based editor module supports 14 commands which operate on plans. These commands are specified using a simple English-like command language. Figure 30 shows a grammar for this command language. Numerous examples of commands are shown in the scenarios in Chapters II & III.

```
command ::= Define a [cliche-name] definition [with parameter-list].
          | Add a parameter variable-name [to definition].
          | Fill role-reference with instance.
          | Replace reference with instance.
          | Insert instance.
          | Copy definition [to definition].
          | Remove reference.
          | Share reference and reference.
          | Highlight reference.
          | Comment [definition].
          | What needs to be done [in definition]?
          | Analyze [definition].
          | Finish editing [definition].
          | Use language SYMBOL.

definition ::= definition-type SYMBOL
definition-type ::= program | function | procedure | package | cliche

parameter-list ::= a parameter SYMBOL | parameters SYMBOLS
SYMBOL ::= a name rendered in upper case
SYMBOLS ::= SYMBOL and SYMBOL | SYMBOL, {SYMBOL,} and SYMBOL

reference ::= role-reference | function-reference
role-reference ::= the [ordinal] role-name {of the role-name} [in definition]
function-reference ::= the [ordinal] SYMBOL [in definition]
role-name ::= the name of a role rendered in lower case
ordinal ::= first | second | third | fourth | fifth | sixth | ... | last

instance ::= CODE
           | a-or-an cliche-name [of instances]
           | role-reference
           | a use of function-reference
CODE ::= an expression in the current language rendered in upper case
a-or-an ::= a | an
cliche-name ::= the name of a cliche rendered in lower case
instances ::= instance
            | instance and instance
            | instance, {instance,} and instance
```

Figure 30: The knowledge-based command language.

References

The most basic part of the command language is the ability to refer to roles in a program. This is done by using the same kind of phrase which is used to define a role. For example, one might say "`the step of the enumerator`".

The cursor position in the Emacs buffer is an implicit part of each command. It is used to disambiguate references. The phrase "`the step of the enumerator`" is interpreted to mean the step of the enumerator in the program containing the cursor.

Alternately, the definition the role is in can be stated explicitly in a reference phrase. For example, one might say "`the enumerator in the program REPORT-TIMINGS`".

If there is more than one role of the same name in a given program, then a role reference can be disambiguated with reference to the execution order of the forms in the program. For example, one might say "`the second enumerator`" in order to refer the the second enumerator (in execution order) in the program

currently being worked on.

One can refer to both empty roles and filled roles. However, the ability to refer to a filled role depends on the fact that the filled role is represented in the plan — i.e., that the program has not been reanalyzed since the role was filled.

In addition to the roles in a program, one can refer directly to the various function calls in a program. This is done by using the name of the function being called. As with roles, one can use ordinal numbers to disambiguate a function reference, and explicitly specify the program which contains the function call. For example one might say "`the last FORMAT in the program SIMPLE-REPORT`".

Instances

Another basic part of the command language is the ability to create plans corresponding to instances of cliches and pieces of code. There are four kinds of instances. An instance can specify a literal piece of code. In this case, the indicated code is analyzed in order to create an equivalent plan.

An instance can also be a role reference or a use of a function reference. In either of these cases, the instance is converted into a trivial plan which represents data flow from the indicated segment in the plan for the program being worked on.

An instance phrase can be an instance of a cliche. In addition to the name of the cliche, a cliche instance can specify sub-instances to be used to fill roles of the cliche. The correspondence between sub-instances and the roles of the cliche is specified by the `PRIMARY-ROLES` declaration of the cliche definition.

When an instance phrase is an instance of a cliche, the plan for the cliche is copied out of the library and the constraints for the cliche are run. These constraints can access any sub-instances which are specified as part of the instance phrase. (Note that this is the only time that `RENAME` constraints are run.) If there are any sub-instances specified for roles of the instantiated cliche then the "`Fill`" command is then called in order to fill the appropriate roles in the instance.

For the most part, the grammar in Figure 30 is an LL0 language and therefore trivial to parse. However, there are two areas of complexity involving instances. First, an instance of a cliche can contain an instance of a cliche as one of its sub-instances. If this cliche instance in turn contains sub-instances, then it may be ambiguous as to which cliche a particular sub-instance belongs to. This problem can be seen in the phrase "`a list of X, a list of Y, Z, and W`" which could be equivalent to either `(LIST X (LIST Y Z W))` or `(LIST X (LIST Y) Z W)`. This problem is arbitrarily resolved by making the lowest level cliche instance contain as much of the command as possible (e.g., choosing the first alternative above).

Second, Parsing the non-terminal ***CODE*** can be very difficult because it requires the ability to parse whatever language KBEmacs is currently operating on. In order to decouple the command parser from the program text parser, restrictions are placed on what code can be typed in a command. Such code is required to be either a string literal, a number, a symbol, an expression surrounded by parentheses or a quoted instance of any of the above. This restriction makes it possible for the command parser to delimit a piece of code without having to actually parse it. (Fortunately, although the restriction above is not as natural for Ada as it is for Lisp, it is still palatable.)

Knowledge-Based Commands

The subsections below describe how each of the commands in Figure 30 is implemented. Each of the commands is implemented as a special purpose procedure which operates directly on the plan formalism. Most of these procedures are straightforward — the plan formalism does most of the work.

Before any command is run, the knowledge-based editor checks to see whether text editing has been applied to the program containing the editing cursor. If it has, then the program text is analyzed in order to create an up to date plan.

Typically, running a command will modify the plan for the program being worked on. When this is the case, the plan is recoded, and the cursor is positioned before the first *interesting* change in the program text. This position is determined by comparing the program text before and after the command under the assumption that changes to the body of a program are more interesting than changes to declarations.

The Knowledge-Based Command "`Define`"

The basic action of the "`Define`" command is to create a plan corresponding to an empty program definition. The command specifies the name of the definition and the type of the definition — i.e., whether it is a function, cliche, procedure, etc. This information is entered into the plan. The command checks that there is not already a program defined with the same name and complains if there is.

Optionally, a "`Define`" command can specify a list of parameter names. If these are specified then they are entered as `LAMBDA-ARG` inputs of the top level segment in the plan.

A "`Define`" command can also specify the name of a cliche which is to be instantiated as the body of the program being defined. This form of the "`Define`" command is merely an abbreviation for a simple "`Define`" command followed by an "`Insert`" command.

The Knowledge-Based Command "`Add a parameter`"

This command makes it possible to add additional parameters to a program definition. This is done by adding more `LAMBDA-ARG` inputs to the top level segment of the plan. If no destination definition is specified then it defaults to the program containing the editing cursor.

The Knowledge-Based Command "`Fill`"

The "`Fill`" command is the central knowledge-based command. It fills a role with an instance. The command first checks to see that the role has not already been filled. If the role has been filled, an error message is generated.

In order to fill the specified role with the specified instance, the role segment is physically replaced by the plan for the instance. The locality and additivity of the plan formalism guarantees that this is all that needs to be done. If a compound role is being filled, then the matrix and sub-roles inside of the compound role are removed before inserting the instance. Thus, the matrix of the instance replaces the matrix of the compound role.

The only real complexity in this is that the "`Fill`" command works hard to link the instance plan into the data flow of the larger plan. This is done using several heuristics. If the instance plan has a return value, then this value is routed to wherever the value of the empty role was used. Similarly, arguments to the instance plan are connected up to the data flow in the larger plan by looking at what the inputs of the empty role were connected up to.

If there are any references to free variables in the instance plan then the "`Fill`" command tries to determine what data flows correspond to these variable names, and link them to the free variable references. Lastly, if there are empty input roles in the instance plan then the "`Fill`" command tries to fill them with data flow by looking at what the inputs of the empty role were connected up to.

Once the instance plan is connected into the larger plan, any constraints specified for the cliche which contains the role which has just been filled are rerun. Note that constraints may be run many times when the "`Fill`" command is called recursively either by the "`Fill`" command or by constraints themselves. (Both `DEFAULT` and `DERIVED` constraints call the "`Fill`" command in order to perform their actions.)

The Knowledge-Based Command "Replace"

The "Replace" command is used to replace a filled role or function call with an instance. It first checks to see that the reference to be replaced is not an empty role. It then removes the contents of the role or function call segment and uses the fill command in order to insert the plan for the instance into the plan.

The Knowledge-Based Command "Insert"

The "Insert" command inserts an instance into a plan at a place corresponding to the position of the editor cursor. To do this, it inserts a special role at the position of the cursor and then analyzes the program in order to get a plan containing that role. It then calls the "Fill" command in order to fill this special role with the specified instance.

Embedded instances of {a ...} annotation are handled exactly the same way. They are converted into "Insert" commands at the time when the program text is analyzed.

The Knowledge-Based Command "Copy"

The "Copy" command is similar to an "Insert" command except that, instead of instantiating the cliche specified as its first argument, it copies the whole cliche verbatim including the cliche declarations. All of this is then inserted into the plan without running any constraints. If no destination definition is specified, then it defaults to the program containing the editing cursor.

The Knowledge-Based Command "Remove"

The "Remove" command deletes the indicated role or function call from the plan. When doing this it also deletes any other segment which exists solely to provide data to the deleted segment. This is done on the theory that all such segments have become useless.

The Knowledge-Based Command "Share"

The "Share" command takes two references and shares them together. The references are first resolved to segments in the plan. Then the plan is checked to see that the two segments compute the same thing. This is relatively straightforward to do by simply comparing the segments and the sources of the data flow to them. The locality of information in the plan formalism greatly facilitates this comparison.

If the two segments do compute the same thing, then whichever segment comes later in the plan is removed and data flow is routed from the first segment to all of the places where the value of the second segment was used.

The Knowledge-Based Command "Highlight"

The "Highlight" command uses the standard editor highlighting mechanism (underlining) to highlight a role or a function call. This command depends on the fact that the plan formalism maintains a correspondence between segments and the code which implements them.

There are two basic limitations to this command. First, there are many situations where the plan formalism fails to be able to record what program text corresponds to a role. Second, the standard editor highlighting mechanism is not capable of highlighting discontiguous sections of the program text. Thus, if a role corresponds to discontiguous pieces of program text (e.g., an enumerator) it cannot be fully highlighted. The latter difficulty would be easy to overcome. The former difficulty might not be.

The Knowledge-Based Command "Comment"

The "Comment" command creates a comment for a program in the form of an outline as shown below. If no particular definition is specified as part of the command, then a comment is constructed for the program containing the cursor. The comment is created by following the CLICHE and CLICHE-ROLE facts in a plan and

the DESCRIBED-ROLES and COMMENT declarations for the cliches in the plan.

```
;;; The function REPORT-TIMINGS is a simple-report.
;;;   The file-name is "report.txt".
;;;   The title is "Report of Reaction Timings (in msec.)".
;;;   The enumerator is a list-enumeration.
;;;      It enumerates the elements of TIMINGS.
;;;   There are no column-headings.
;;;   The print-item is a tabularized-print-out.
;;;      It prints out (CAR LIST) in columns.
;;;   The summary is an idiosyncratic computation.
```

The first line of the comment specifies the name of the function/procedure being described and the top level cliche in its plan. The subsequent lines in the comment describe the DESCRIBED-ROLES of the top level cliche in order. Each role is described in one of four ways. If the role is missing this fact is reported. If the role is filled with a cliche then the name of this cliche is given and a one line description of the filling cliche is created based on the COMMENT declaration for the filling cliche. If the role is filled with a simple non-cliched piece of code then this piece of code is exhibited. Finally, if the role is filled with a complex piece of non-cliched computation then the phrase "idiosyncratic computation" is used to describe the role.

The above approach to constructing a comment is based on the earlier work of Cyphers [Cyphers 82] which in turn descends from the work of Frank [Frank 80]. Both of these efforts investigated the basic issue of generating an English language description of a program based on the information in the plan formalism. Frank focused on describing the basic properties of programs which were not constructed using cliches. Cyphers switched the focus to generating an explanation of a program based on the cliches used to construct it. He attempted to create free flowing English paragraphs describing a program in full detail.

The current system uses Cyphers' basic method modified in two ways. First, most of the problems associated with generating free form English are side stepped by outputting the comment in an outline form. Second, the complexity of the comment is limited by truncating the description at a nesting depth of two.

The Knowledge-Based Command "What needs to be done"

This command looks at the plan for the specified definition and determines what roles still need to be filled in and what output roles still need to be used. A report of this information is displayed to the programmer in a temporary window at the top of the screen. If no definition is specified as part of the command, then a report is displayed of what needs to be done for every program for which a plan exists.

The Knowledge-Based Command "Analyze"

The "Analyze" command triggers the analysis of the named definition if it has been textually modified since it was last analyzed. If no definition is specified, then the definition containing the editing cursor is analyzed.

The Knowledge-Based Command "Finish editing"

This command is identical to the "Analyze" command except that it calls the "What needs to be done" command in order to check that the program is indeed finished and then removes any output role annotation from the program text so that it can be processed by a standard interpreter or compiler.

There is also a procedural hook which is part of the "Finish editing" command that checks to see whether any additional actions should be performed based on the program just completed. This is how the creation of special cliches after the definition of the package MAINTENANCE_FILES in Chapter III is triggered.

The Knowledge-Based Command "`Use language`"

This command sets an internal variable which controls what language is being used for output to the programmer. It is assumed that the programmer will use this same language when typing literal pieces of code in commands.

The commands `s-N` and `s-P`

The commands `s-N` (move to the next empty role) and `s-P` (move to the previous empty role) are not actually supported by the knowledge-based editor. Rather they are implemented as user-defined commands in Emacs. They operate in a purely textual manner simply searching for the next (previous) occurrence of an empty piece of role annotation. They are described here because they fit in logically with the knowledge-based command language since they are based on the concept of roles.

Possible Improvements

The knowledge-based editor is one of the most experimental parts of KBEmacs and is still undergoing rapid evolution. It is expected that numerous improvements will be made as experimentation continues. However, as demonstrated by the scenarios in Chapters II & III, the current commands seem to be quite useful.

Language Independence

The knowledge-based editor commands are almost completely programming language independent since they operate almost exclusively in the domain of plans. The only exceptions to this are the "`Comment`" and "`Highlight`" commands which have to interact with the actual language being used. Even these commands are largely programming language independent. They both depend on the fact that the coder has already created program text in the appropriate language.

Interface

An interface (implemented by Pitman [Pitman 83]) unifies ordinary program editing and knowledge-based editing so that they can both be conveniently accessed through the standard Emacs-style Lisp Machine editor.

The Style of the Interface

There are two major aspects to the style of the interface: the way the programmer types knowledge-based commands and the way the results of commands are communicated to the user. The command interface is strongly integrated with Emacs in both operation and style. The knowledge-based commands are supported as an extension of the Emacs command set. More than this, as discussed below, the `s-X` window which is used for typing knowledge-based commands operates in very close analogy with the `m-X` window which is used for typing extended Emacs commands. In addition, a number of single character commands such as `s-N` and `s-;` are provided because it is in the style of Emacs to do so.

The results of knowledge-based commands are communicated to the programmer by altering the program text in the Emacs buffer. The intention is that the programmer should interact with KBEmacs much in the same way that he might interact with another programmer typing at the same keyboard. There are four basic virtues to this approach. First, program code provides a detailed unambiguous description of the net effect of the set of knowledge-based commands used so far, expressed in a language which is familiar to the programmer. Second, using program text facilitates the intermixing of direct editing with knowledge-based editing. Third, using program text submerges plans into the background allowing the programmer to think textually much of the time. Fourth, since the developing program is always represented in the Emacs buffer, the programmer can apply any of the standard Lisp Machine tools to it at any time.

The above not withstanding, it is possible that there are other things which might be better than program text as a vehicle for communication between KBEmacs and the programmer. Program text has the problem that it can be quite large. A 100 line program is still hard to understand even if it was created using only 10 knowledge-based commands. It would be desirable to have some more compact representation.

One thing which has been considered but not yet explored in detail is using a stylized comment like the one created by the "`Comment`" command as a continually updated summary of the program. The goal would be to allow the programmer as well as KBEmacs to edit this comment. Given that the analyzer cannot currently recognize what cliches could have been used to create a program, it would not currently be possible to create such comments after direct editing had been applied. Nevertheless, such an interface might be a very useful extension to KBEmacs.

Typing Knowledge-Based Commands

As mentioned above, the `s-X` window which is used for typing knowledge-based commands operates in very much the same way as the `m-X` window which is used for typing extended Emacs commands. The interaction occurs in a small window below the editor buffer. Once one or more commands are typed, they are executed by typing the Lisp Machine Key `<end>`. (In the screens in scenarios in Chapters II & III the commands are shown above the editor buffer because this improves the visual flow of the scenarios.)

An interesting feature of the `s-X` window is that it supports word completion. This significantly reduces the number of characters which actually have to be typed in order to type a command. In the example below, bold face italics is used to indicate the characters which are introduced via word completion as opposed to the characters typed by the user.

s-X F***ill*** the enum***erator with*** a list-enum***eration*** of TIMINGS. <end>

Another aspect of the `s-X` window is that the full set of Emacs commands can be applied to it. As a result, the user can easily correct and modify the command he is typing.

The typing of knowledge-based commands is further facilitated by the existence of a number of abbreviated commands such as `s-F`. These commands (summarized in Figure 31) operate by initializing the `s-X` window with a partially typed command. For example, if the editing cursor happened to be positioned before the enumerator, then the command above could be typed as shown below, further reducing what the user has to type.

s-F ***Fill the enumerator with*** a list-enum***eration*** of TIMINGS. <end>

Some abbreviated commands (e.g., `s-;`) automatically generate everything which has to be typed for a command.

```
command         s-X buffer initialization
s-F     <==> Fill the next empty role with
s-R     <==> Remove the next empty role
s-;     <==> Comment the current definition. <end>
s-W     <==> What needs to be done? <end>
s-A     <==> Analyze the current definition. <end>
s-<end> <==> Finish editing the current definition. <end>
```

Figure 31: Summary of abbreviated commands.

There are two key potential problems with any English-like command language. First, such command languages are often wordy and cumbersome to type. Here, that does not seem to be very much of a problem because various mechanisms reduce what needs to be typed to the point where it is not clear that any command interface could reduce it much further.

Second, with an English-like interface there is always the danger that a user will make the mistake of assuming that the interface language actually is English and start typing all kinds of sentences the system cannot understand. Here, this problem is ameliorated by the fact that the command language is so restricted that it is easy for the user to get a good grasp of what he can type and what he cannot.

Asynchronous Processing of Knowledge-Based Commands

An interesting aspect of the interface not shown in the scenarios is that it contains a mechanism for dealing with the slowness of KBEmacs. The command `s-X` described above forces the programmer to wait and do nothing until the specified knowledge-based commands have been processed. An additional command `h-X` (typed by holding down the Lisp Machine keyboard shift key `HYPER` and typing an `X`) is identical to the `s-X` command in every way, except that it causes the specified knowledge-based commands to be processed asynchronously in a background process.

While this asynchronous processing is going on, the programmer may continue to work on other things. (Problems will of course arise if the programmer attempts to work on the same program the knowledge-based editor is working on.) Although it takes a little getting used to, it turns out that asynchronous processing does go a long way towards making the slowness of KBEmacs more palatable.

The programmer is given a signal when the asynchronous processing is completed. However, the results of the commands are not immediately inserted into the editor buffer. Rather, the knowledge-based editor waits until the programmer requests that the changes be inserted by typing a `h-<resume>` command. (It was discovered through experimentation that inserting the changes immediately can be very disconcerting.)

The asynchronous processing above is supported by a message passing mechanism which controls the interaction of a number of independent processes. This message passing mechanism (which was a major focus of Pitman's work) was generalized to the point where several Lisp Machines could be used at once in order to increase the throughput of KBEmacs.

Implementation

For the most part, the implementation of the interface is straightforward. However, it is complicated by the fact that it has to interact directly with the undocumented internal functions of the standard Lisp Machine editor. Nevertheless the interface is a relatively well tested and robust part of KBEmacs.

Language Independence

The interface itself is totally language independent. However, it relies on the fact that the standard editor has a basic understanding of the language being edited. If this is not the case, then the standard editor has to be extended.

For example, version 4 of the Lisp Machine editor does not support Ada. In order to make the interface work for Ada, a few basic extensions had to be made in order to provide minimal support for Ada. In particular, the editor was given the ability to locate Ada programs in a buffer. (This also provided support for the standard Emacs commands `m-.` (goto definition) and `c-shift-C` (compile definition).) However, support was not provided for highlighting parts of an Ada program. As a result, the "`highlight`" command does not work for Ada even though the knowledge-based editor knows what should be highlighted.

Syntax-based editing of programs is not supported directly by KBEmacs but rather by the standard editor. As a result, syntax-based editing will be supported only if the standard editor supports it. It is in principle possible to extend the standard editor to support syntax-based editing of a new language, but it is not particularly easy.

The Lisp Machine editor has not yet been extended to support syntax-based editing of Ada programs. As a result, only text-based editing can be applied.

Like syntax-based editing, interpretation and/or compilation of programs is not supported directly by KBEmacs but rather by the standard Lisp Machine software. If interpretation and/or compilation is desired for a new language, then an interpreter and/or a compiler will have to be implemented.

A simple Ada interpreter has been implemented so that the programmer can test Ada programs. This interpreter is capable of running the programs in Chapter III. However, it only supports part of the language and is only intended as an illustration.

Going From a Demonstration To a Prototype

As discussed in Chapter IV, in order to convert KBEmacs from a demonstration system into a prototype, the system needs to be reimplemented with special attention to speed, robustness, and completeness. This section suggests several changes which should be made were the system to be reimplemented. It also describes an experiment which suggests that the system can be straightforwardly speeded up by more than an order of magnitude.

A More Efficient Representation For Plans

Figure 32 shows an experimental plan implementation which is more efficient than the current implementation of the plan formalism.

```
new-seg ::= (name: name
           supseg: [new-seg]              subsegs: ({new-seg})
    pbm-role-name: [ACTION | PRED | JOIN | OP | REC | INIT]
        plan-type: LAMBDA |  EXPR | XOR | PRED | SSR | TEMPCOMP |
                   JOIN | CONSTANT | FUNCALL | RECURSION
   plan-type-info: name-or-constant
      role-supseg: [new-seg]          role-subsegs: ({new-seg})
        role-name: [name]
        role-type: INPUT | OUTPUT | INTERNAL
      cliche-name: [name]
       rec-supseg: [new-seg]           rec-subsegs: ({new-seg})
             code: ({form})
         def-type: [FUNCTION | PROCEDURE | PACKAGE | CLICHE]
         def-name: [name]
     declarations: ({cliche-declaration})
            cases: ({new-case}))

new-case ::= (name: name
              seg: new-seg
             side: INPUT | OUTPUT
             when: [form]
       cflow-from: ({new-case})            cflow-to: ({new-case})
     is-lower-env: ({new-port})
            ports: ({new-port}))

new-port ::= (name: name
             case: new-case
       dflow-from: ({new-port})            dflow-to: ({new-port})
             kind: ARG | FREE-VAR | RETURNVAL | SIDE-EFFECT | DUMMY
        kind-info: [argno | name]
         var-name: [name]
        lower-env: [new-case]
          part-of: ({new-port})           has-parts: ({new-port})
     rec-superior: [new-port]         rec-inferiors: ({new-port})
 side-effect-outs: ({new-port})      side-effect-in: [new-port])
```

Figure 32: A More Efficient Representation For Plans.

The experimental implementation encodes basically the same information as the implementation

discussed in the beginning of this chapter. However, it represents this information much more compactly. The experimental implementation represents plans in terms of record structures which point directly to each other, instead of as a set of facts in a data base. There are three kinds of record structures: one for each segment, one for each case of a segment, and one for each port in a case.

In the experimental data base, retrieval is done by directly following pointers instead of by means of patterns and indexing. (Note that all of the pointer fields are doubly linked. For example, each segment points to its super-segment and to each of its subsegments.) Following pointers directly speeds up retrieval by many orders of magnitude. In addition, since all of the information about a segment or port is contained in a single structure, only one retrieval has to be performed in order to get all of this information. In the current data base, several retrievals have to be performed to get all of the information about a segment or port.

In addition to representing information more compactly, the experimental plan representation in Figure 32 streamlines and improves the plan formalism in several ways.

In the plan formalism, segcases (as opposed to segments) appear in all of the facts even though they are seldom needed. In the experimental plans, cases are only used to fix the position of ports, and as the end points for control flow. Another difficulty with cases in the plan formalism is that, in isolation, it is not possible to determine whether a case is attached to the input or the output side of a segment. It turns out that this makes a number of operations on plans needlessly complex. Therefore, the side of a case is made explicit in the experimental plans.

The segkind, segtype, and plantype fields are largely redundant in the plan formalism. They are combined into a single field in the experimental plans. Similarly, the distinction between lambda-args and actual-args is not important and is eliminated.

In the plan formalism, there are several situations where connections between ports which are closely analogous to data flow are not represented by data flow. In particular, both the connections between ports on a join and the connection between a temporal port and its lower-env are represented by special purpose facts. This makes it needlessly complex to trace data flow through a plan. In the experimental plans, data flow is used to represent all data connections.

In the experimental plans, temporal ports are the same as non-temporal ports except that they have a lower-env specified. Whether a port is an input or an output is determined by looking at the side of the case it is in. The experimental plans have a special field for specifying the correspondence between the ports of a segment and the ports of any recursive instances of that segment. In the plan formalism this information is awkwardly encoded by requiring that corresponding ports must have the same name.

In the experimental plans, the plan building methods are simplified. In particular, SSRs are simplified by no longer allowing them to have an initialization. (The effect of an initialization can be achieved by having a TEMPCOMP which groups the initialization and SSR together.) Removing initializations from SSRs makes it possible to get rid of the special body segment as well. This converts SSRs into one level structures instead of two level structures. With this change, all plan building methods become one level structures and plan building methods can simply be understood as specifying how the subsegments of a segment interact. As an additional simplification, all of the subsegments of a TEMPCOMP are referred to as ACTIONs getting rid of the fancy names ENUM, FILTER, etc.

Gregory Faust and the author performed an experiment to demonstrate the advantages of the experimental plan representation shown in Figure 32. The representation was implemented and parts of the knowledge-based editor were reimplemented in terms of the experimental representation. The reimplemented knowledge-based editor was then tested on the task of instantiating a relatively complex cliche and inserting it into a plan. This resulted in an increase of speed by a factor of 30 when compared with the current implementation of KBEmacs. There is reason to believe that the experimental plan representation

would lead to at least an order of magnitude increase in speed in the other parts of KBEmacs as well.

Eliminating Temporal Decomposition

The introduction of the improved plan representation presented above would speed up KBEmacs without fundamentally changing the way the system operates. There are several ways in which it might be possible to significantly simplify and thereby speed up the system by changing the way it operates.

One thing which could be done would be to remove temporal decomposition from the analyzer and temporal composition from the coder. As will be discussed below there are two ways in which this could be done without reducing the capabilities of the system. If this were done, it might speed up the system by as much as a factor of 2. It would also significantly reduce the complexity of the analyzer and coder.

Perhaps the best way to remove the idea of temporal composition/decomposition from KBEmacs would be to incorporate it into the target language. This could be done by using a notation and compiler extension (such as the one presented in [Waters 83a,84a] and discussed in the next chapter) which directly supports computations in terms of series of values. KBEmacs could manipulate these computations as simple expressions, while the compiler extension handled the problem of temporal composition.

Another approach would be to simply represent all loops as loops and embed knowledge about temporal composition in the knowledge-based editor. This would increase the complexity of the knowledge-based editor, but only when it was actually engaged in inserting a cliche into a loop. In any case, the analyzer and coder would always be able to operate on temporally composed plans.

Eliminating Plan Building Methods

Another way in which KBEmacs could be fundamentally simplified would be to eliminate the idea of plan building methods. As has been discussed in various places above, plan building methods are used for three things in the current system. First, they make it possible to create a grouped plan even though cliche recognition is not supported. Second, the grouped plan makes it possible for KBEmacs to make an intelligent guess about what part of the matrix of a cliche is associated with a compound role. Third, the grouped plan is used by the coder as a guide for what control constructs to use in the code produced.

As discussed in the section on plans above, although it was originally thought that grouped plans were essential as a basis for interaction with the programmer, this has not turned out to be the case. In addition, it would probably be relatively straightforward to determine what part of the matrix of a cliche is associated with a compound role without using grouping. This could be done by either creating a new kind of annotation or by taking advantage of the fact that only a very few special kinds of compound roles are actually used in the current cliche library. (It should be noted that if computation in terms of series were supported in the target language then there would not be any compound roles at all in the current cliche library.)

As a result, the only essential use of plan building methods is in the coder. First of all, this suggests that the process of grouping should be moved to the coder. This would significantly simplify the rest of the system. Unfortunately, confining grouping to the coder might not speed things up since the plan would have to be regrouped every time it was coded. However, it might be possible to simplify the grouping process if it were aimed only at satisfying the needs of the coder. In particular, there would be not need to actually encode the grouping in the plan. It could be recorded in temporary structures used only by the coder.

Knowledge About Data Structures

In order to become a true prototype, KBEmacs would have to be extended so that it had a better understanding of data types. There are two aspects to this. First, the plan formalism would have to be extended so that it was capable of representing the fact that most data types can be determined by static analysis. This is even true in Lisp. The current lack of information about data types in a plan reflects the

pessimistic Lisp assumption that nothing can be determined about data types by static analysis.

Second, in order to really support a language like Ada which allows extensive user definition of new data structures the plan formalism would have to be further extended so that it could represent how complex data structures are built out of simple ones. The work on the plan calculus suggests how this could be done.

Defining a Realistic Cliche Library

A final step in converting KBEmacs into a true prototype would be the construction of a much more complete cliche library. This might require the definition of as many as a thousand cliches. Although this would not be an easy task, there is no theoretical reason why it could not be done. (It should be noted that, the speed of KBEmacs is independent of the size of the cliche library.) If a large body of cliches were defined, then it would be beneficial to add some kind of indexing facility into the library in order to facilitate the retrieval of cliches.

VI - Future Directions

Work in the PA project is continuing in several areas. The largest amount of effort is going into the construction of the next demonstration system. As discussed extensively in Chapter IV, this new system will go beyond KBEmacs primarily by supporting contradiction detection and interaction in terms of design decisions rather than specific algorithms.

Applications of the Current Techonology

Another area of activity centers around applications of some of the ideas behind KBEmacs. In particular, several systems have been built which are much more modest in their goals than KBEmacs, but which have the advantage of being fully practical prototypes. This work includes both efforts within the PA project and work by other groups.

Tempest

A particularly interesting application of the ideas behind KBEmacs is the Tempest editor implemented by Sterpe [Sterpe 85]. At the highest level of abstraction, Tempest is very much the same as KBEmacs. It has a user extendable library of cliches referred to as *templates* which contain roles referred to as *blocks*. Users can instantiate templates and fill in blocks. Simple constraints can be used to control the way blocks are filled in. Tempest maintains a record of the structure of the text being edited in terms of templates and blocks. Several commands make use of this structure — e.g., an instance of a template can be replaced by an instance of another template. Tempest is built on top of a standard Emacs-style text editor. Text-based editing commands can be freely intermixed with template-based editing commands at any time.

The primary difference between Tempest and KBEmacs is that, instead of using a plan-like representation, Tempest operates almost entirely in terms of text. For example, the process of instantiating a template boils down to merely inserting a file containing the text of the template into the editor buffer. The only non-textual effect of instantiating a template is that it causes Tempest to update its record of the structure of the text being edited.

The textual orientation of Tempest makes the system vastly simpler than KBEmacs. For example, there is no analyzer module and no coder module. In consequence, Tempest is three to four orders of magnitude faster than KBEmacs. This enables Tempest to run with acceptable speed on an IBM PC.

On the negative side, the textual orientation of Tempest fundamentally limits the power of the system. The problem is that Tempest has no understanding of the text it is operating on. One result of this is that the system is very limited in what it can do with a template — it can insert a template textually into the editor buffer, but it cannot modify it so that it will fit in better with the surrounding text.

In principle, Tempest has a very broad range of applicability — it can be used to edit anything which can be represented as text. However, in practice, it is only really useful in situations where templates can be combined textually without having to be modified. For example, Tempest is not particularly useful for the construction of programs. On the other hand, Tempest can be very useful for document preparation.

The scenario in [Sterpe 85] shows Tempest being used to construct a document file which is to be processed by a text formatting program. This is an application which fits together particularly well with Tempest's strengths and weaknesses. In the scenario, Tempest is used to build a document file out of templates and other chunks of text. The text formatter takes care of fitting the chunks of text together aesthetically.

An interesting feature of Tempest is that it goes beyond KBEmacs in several ways. In particular, it provides improved facilities for dealing with textual editing, specifying structure, and flexible viewing of the text being edited.

Unlike KBEmacs, Tempest is able to maintain its record of the structure of the text being edited even after arbitrary text-based editing by the user. This is done by incrementally updating the representation of structure as part of every text-based editing request.

Tempest provides a set of commands which enable the user to impose arbitrary structure on preexisting text. In particular, blocks can be gathered together into a hierarchy of *groups*. A block can be included in several different groups. This enables several different viewpoints on the structure of the text to be represented at once.

KBEmacs provides two commands which support flexible viewing of the text being edited. The first command displays an outline showing the structure of the text being edited. (This is in many ways similar to the KBEmacs command "Comment".) The second command constructs a special buffer which displays a single group from the text. Typically, this group will be a non-contiguous collection of blocks in the text. An interesting aspect of this command is that any editing done in the special display buffer carries over to the main text.

Efficient Computation With Series

An interesting application of one of the ideas behind KBEmacs is the programming language extension described in [Waters 83a] in terms of Lisp and in [Waters 84a] in terms of Ada. This extension is inspired by the way loops are represented in the plan formalism. The extension makes it possible to express computations as compositions of functions on series of values and uses some of the same algorithms which are used by the coder module of KBEmacs in order to compile the computations into efficient iterative loops.

An improved version of this language extension is currently under development. The improved version is supported by a special binding macro S:LET*. The code below shows how S:LET* could be used to implement the program MEAN-AND-DEVIATION which was used as an example in Chapter II.

```
(DEFUN MEAN-AND-DEVIATION (TIMINGS)
  (S:LET* ((TIMING (ENUM-LIST TIMINGS))
           (COUNT (ACC-COUNT TIMING))
           (MEAN (// (ACC-SUM TIMING) COUNT)))
    (LIST MEAN (- (// (ACC-SUM (^ TIMING 2)) COUNT) (^ MEAN 2)))))
```

Series functions such as ENUM-LIST, ACC-COUNT, and ACC-SUM generate and consume series of values. ENUM-LIST enumerates a list, generating a series of the elements in the list. ACC-COUNT accumulates a count of the number of values in a series. ACC-SUM accumulates a sum of the numbers in a series. A library of standard series functions is provided along with mechanisms for defining additional ones.

The macro S:LET* is exactly the same as the macro LET* except that it supports the binding of variables to series of values. For example, the variable TIMING contains the series of elements in the list TIMINGS.

When a non-series function is applied to a series of values, then it is automatically applied to each element in the series creating a new series. For example, the expression (^ TIMING 2) creates a series of the squares of the numbers in the series TIMING.

The macro S:LET* makes it possible to state a variety of algorithms in a compact functional style. However, these algorithms could be represented in essentially the same way by using functions which operate on lists of values. The key contribution of S:LET* is that it makes it possible to state a variety of algorithms in a compact functional style without suffering a reduction in efficiency in comparison with iterative loops.

Psychological Validation

Several researchers outside of the PA project have been able to make use of some of the ideas in KBEmacs. One example of this is the work done by Soloway. In a number of experiments Soloway has investigated how programmers think about programs.

Soloway has been able to show that, in many situations, the kinds of cliches used by KBEmacs do in fact correspond to the way programmers think. For example, one set of experiments [Soloway 84] shows that programmers use algorithmic fragments similar to the loop cliches used by KBEmacs when thinking about loops.

An interesting application of this research is that it makes recommendations about what language features are helpful to programmers. In particular, it suggests that language features are helpful when they can easily express the cliches used by programmers and unhelpful when they make it difficult to express these cliches. For example, [Soloway 84] presents experiments which show that languages which require all loops to be written in terms of `DO WHILE` are unsatisfactory because they make it difficult to state many common cliches and therefore lead to unnecessary bugs.

Assisting Novice Programmers

Both Soloway [Soloway 83] and Eisenstadt [Laubsch 81] have used the concepts of plans and cliches in order to implement systems which can assist novice programmers. Their systems inspect programs written by novice programmers, attempting to determine what bugs exist in the programs and give advice on how these bugs might be fixed.

Both systems are designed to be used in a tutorial setting where the exact program to be written is known in advance. The systems depend on having a detailed description of exactly how the correct program is supposed to work and descriptions of errors which novice programmers commonly make. These descriptions are created manually.

Student programs are matched against the description of the ideal program and any differences detected are reported as bugs. Aspects of the plan formalism and symbolic evaluation are used to create quasi-canonical summaries of both the ideal program and the student programs in order to reduce the incidence of false mismatches.

The Intelligent Program Editor

Another application of PA ideas outside of the PA project is the Intelligent Program Editor (IPE) currently being developed by Shapiro [Shapiro 83]. A primary motivation behind IPE is the observation that comprehending what a given program does is the hardest part of program modification. The parts of IPE which have been implemented so far address this problem by assisting a programmer to locate relevant portions of a program.

In addition to program text, IPE maintains a data base of information about the program called the *extended program model*. Among other things, this data base contains plan-like information about the logical structure of the program and about the cliches which are used in the program. This and other information is derived by various program analysis tools.

As of [Shapiro 83], the principal component of IPE is a navigation tool which helps a programmer locate portions of a program based on semantic criteria. Using this tool a programmer can interactively zero in on the portion of a program he wishes to see. When doing this he can refer both to semantic information in the extended program model and to syntactic features of the program. For example, the programmer might ask to see any summation loops involving the variable `TEST-SCORES`. IPE would then highlight any loops which semantically corresponded to summation and syntactically referenced the variable `TEST-SCORES`.

Attacking Other Parts of the Programming Process

A final area of activity in the PA project centers around expanding the scope of the PA beyond program implementation. Several pilot studies have already been done investigating how the ideas behind the PA can be applied to other phases of the programming process.

Testing

Chapman [Chapman 82] built a tool (the Testing Assistant) which assists with program testing. This tool keeps track of the test cases which are used in conjunction with a collection of programs and automatically reruns an appropriate subset of these tests whenever any of the programs is modified.

Preparatory to using the Testing Assistant, the programmer divides the collection of programs up into groups. He then specifies which features of the collection of programs are supported by each group.

Once the groups have been specified the programmer proceeds to test the programs by interactively running various test cases. Each time the programmer runs a test case, he specifies which features of the collection of programs are being tested. The Testing Assistant stores the test case in a data base indexed by the features it tests.

Whenever the programmer modifies a program (e.g., in order to fix a bug which is discovered) the Testing Assistant automatically reruns all relevant test cases. A test case is deemed relevant if it tests any of the features which are supported by the group which contains the modified program. (By greatly reducing the number of test cases which have to be run, the relevance test significantly increases the speed of the Testing Assistant.)

An interesting problem arises when a program is modified in such a way that the test cases which directly call it can no longer be run — for example when the number of arguments to the program are changed. The Testing Assistant is able to partly overcome this problem by applying a number of heuristics which specify how to change a test case when a procedure is changed.

As currently constituted, the Testing Assistant is designed to fit in with the kind of highly interactive testing which is typically done in an environment like the Lisp Machine. However, the basic concept could equally well be applied in a less interactive situation. For example, suppose that a set of test cases were developed by a separate testing organization rather than by the programmer. It would still be beneficial to enter them in a data base like the one supported by the Testing Assistant in order to facilitate continued testing when programs are modified.

Debugging

Shapiro [Shapiro 81] demonstrated a tool (called Sniffer) which helps a programmer to debug a program. This tool supports two basic capabilities. First, the Lisp evaluator is modified so that Sniffer can keep track of all of the side-effects which are performed by a program which is being executed. This makes it possible for Sniffer to reverse the execution of a program and return to any previous execution state, no matter what operations the program has performed. Second, Sniffer has a knowledge base of descriptions of particular kinds of bugs which are likely to occur. When an error is detected, Sniffer uses this knowledge in order to try to identify the bug which caused the error.

The most interesting part of Sniffer is the way it identifies bugs. The process begins when an error is detected. An error is detected either when the Lisp interpreter goes into an error state (e.g., when division by zero is attempted) or when the programmer notices that an incorrect output has been produced. The error detection step culminates in the production of a simple description of the manifestation of the error.

Each bug description in Sniffer's knowledge base contains information about three things: the error manifestations associated with the bug, the sequence of execution steps which occur when the bug happens, and a plan for the specific buggy algorithm. Sniffer uses the analyzer module of KBE in order to create plans

for the program being executed and then begins to look for bugs. It is guided in its search by looking at how the error was manifested, and at what evaluation steps actually occurred.

If Sniffer succeeds in finding a part of the program which matches one of the bugs it knows about and which could have led to the error which was detected, then it prints out a bug report. This report describes the bug and suggests how it might be fixed.

Sniffer is merely a demonstration system. In particular, it only knows about a couple of bugs. However, it shows the potential leverage which a knowledge-based approach has on the debugging task.

Documentation

Zelinka [Zelinka 83] proposed a tool which can assist with the task of maintaining documentation. The focus of this tool is understanding how changes in a program are related to changes in documentation.

There are two major underpinnings of the tool. First, each part of the documentation is linked to the parts of the program which are relevant to it. Second, there is a taxonomy of various types of ways in which programs can be modified. This taxonomy is important because particular types of program modifications suggest particular ways in which the documentation has to be modified. For example, if a modification merely fixes a bug, then the documentation should not have to be modified at all.

Every time the program is modified, the tool looks at the links between the modified parts of the program and the documentation and, based on the type of modification, directs the programmer's attention to the parts of the documentation which should be changed. The goal is to reduce the probability that the programmer will overlook parts of the documentation which are relevant to his modification.

The tool is not capable of modifying the documentation itself because it has no understanding of either the documentation or the program. The obvious next step would be to give the tool some understanding of the documentation. However, in order to avoid having to understand free form English, this would require recasting the documentation into a simplified pseudo-English form.

Rapid Prototyping

[Rich & Waters 82] speculate on how the PA could assist with rapid prototyping. The basic idea here is to build up a program very quickly by using cliches which omit a lot of details (such as error checking) and then substitute more complete cliches when it becomes time to construct a full program.

In order to support this kind of rapid prototyping, the cliches in the cliche library are arranged into clusters. Each cluster is centered around a simplified cliche which performs a given algorithm but which makes lots of assumptions and does no error checking. Various combinations of efficiency, robustness, generality, and error checking features are supported by other cliches in the cluster.

A program is initially constructed by using only the simplified cliches. This yields a simplified program which can be tested in simple situations. Once experimentation at the simplified level has been completed, the prototype program is converted into a full implementation.

Conversion is performed semi-automatically. The programmer selects what features he wants the full program to have. The PA then replaces each simplified cliche with another cliche from the same cluster which has the required features. Often, using one of the replacement cliches forces the programmer to supply additional information, because it brings up design issues which did not have to be faced in the simplified program. This is why the conversion process is only *semi*-automatic.

Translation

A particularly interesting pilot study involved program translation. Faust [Faust 81] built a demonstration system which is able to translate Cobol programs into Hibol [Ruth 81] programs.

The translation task performed by this system is particularly difficult for two reasons. First, the system takes great pains to produce output which is not merely correct, but aesthetic. Second, the translation task is rendered inherently difficult by the fact that Hibol is a much higher level language than Cobol. Hibol is a very high level business data processing language. It is a non-procedural single assignment language which uses direct operations on series of values (called *flows*) in lieu of loops.

The Cobol to Hibol translator operates as follows. A source Cobol program is converted into a plan by the analyzer module of the KBE. As the first step of this, the program is parsed by a Cobol parser which was implemented by Glen Burke.

The plan is then analyzed in order to obtain summary information describing what computation the program is performing. This analysis is done in two ways. Recognition is applied in order to identify what kinds of looping fragments are present in the program. A symbolic evaluation process is used in order to construct algebraic equations which describe the net effect of the Cobol program.

The body of a Hibol program is then produced based on the algebraic equations constructed from the plan. As part of this, an algebraic simplification module is used to improve the readability of the Hibol program. The data declaration part of the Hibol program is constructed from the data declaration part of the Cobol program by means of a more or less separate process. This process is relatively straightforward since data declarations in the two languages are relatively similar in form.

As implemented, the Cobol to Hibol translator demonstrates quite satisfactory results. However, the class of programs which can be successfully translated is severely limited by the fact that the translator does not contain a generalized recognition facility which can support plan analysis. Rather, specific procedures must be written to support analysis.

Once a general cliche recognizer (such as the one being constructed as part of the next demonstration system) is available, a more general purpose and powerful translation method will be possible. As shown in Figure 33, this method proceeds in four steps.

```
                                   abstract cliche plan
                                        ↗           \
                    ABSTRACTION /                    \ SPECIALIZATION
                                 /                     ↘
                      Cobol cliche plan          Hibol cliche plan
                                ↗                        \
       ANALYSIS/RECOGNITION /                             \ CODING
                             /                              ↘
                      Cobol text                       Hibol text
```

Figure 33: Translation based on cliches.

First, an analyzer is used to convert a Cobol program into a plan and recognition is used to determine what Cobol cliches could have been used to build the program. Second, *abstraction* is applied to restate these Cobol cliches in terms of abstract cliches which are neutral between the languages Cobol and Hibol. Third, *specialization* selects Hibol cliches which correspond to the abstract cliches derived from the Cobol program. Both of the last two steps are relatively straightforward because they are driven by specialization links between cliches recorded in a cliche library. Fourth, a coder module is used to create aesthetic Hibol code based on the Hibol cliche plan.

Compilation

Translation based on cliches as described above could be used to translate between any two languages. Duffey [Duffey 80] proposed a compiler (called Cobbler) which uses this basic method in order to compile Pascal programs into PDP-11 machine code.

At first glance, the problems associated with compiling Pascal do not seem to be very similar to the problems associated with translating Cobol to Hibol. After all, the goal of the former is efficiency of low level output while the goal of the latter is readability of high level output.

However, the two problems really have a great deal in common. Stated generally, the problem is that the criteria which govern the source are very different from the criteria which govern the output. In order to have the freedom to do a good job of satisfying the output criteria, the input must be restated in a way which frees it from the constraints of the input criteria.

For example, Pascal is heavily biased toward loops which count up. (The special forms of the looping constructs do not support counting down.) In contrast, the most efficient PDP-11 looping instructions only support counting down. As a result of this mismatch, typical Pascal compilers essentially never use the most efficient PDP-11 looping instructions in the code they produce.

The first thing Cobbler does when presented with a Pascal loop is attempt to determine whether the loop counts up because the algorithm requires it to or merely because the algorithm does not require that the loop count down. Cobbler actually goes way beyond this. It abstracts away from the Pascal program and determines the net effect of the loops in the program on various vectors and arrays. It does this so that code generation will not be be constrained by the exact placement of loops in the source program any more than by whether the loops count up or down.

Once an abstract statement of the algorithm being performed is determined, Cobbler proceeds to respecialize the algorithm, making heavy use of the efficiency features of the PDP-11 instruction set. Cobbler produces better code than other compilers because it is able to take a very global view of the algorithm being compiled.

Requirements Analysis

Currently, work has begun on the design of a Requirements Analyst's Apprentice (RAAP). This research (by Howard Reubenstein) is only in its early formative stages. However, the basic outline of the system seems clear.

RAAP will have a data base which describes the requirement being worked on. The requirement will be represented either by a plan-like representation or by a more semantic network-like representation.

RAAP will be able to assist a requirements analyst in several ways. It will be able to receive information in a relatively unordered way and assemble it into a requirement. It will be able to detect some kinds of inconsistencies and incompleteness in the evolving requirement. It will be able to create a document describing the requirement as a whole, and various summaries of parts of the requirement.

The heart of RAAP will be a library of cliches in the domain of requirements. These cliches will be the medium of communication between RAAP and a requirements analyst. It is expected that a typical requirement constructed using RAAP will be composed 90 percent of cliches and 10 percent of new concepts which have to be described to RAAP in detail. The great importance of cliches can be seen from the fact that, if constructing requirements is anything like constructing programs, the largest part of the time required to construct a requirement will be taken up describing the 10 percent of the requirement which is not cliched.

The Programmer's Apprentice

The long term goal of the PA project is to produce a true Programmer's Apprentice — a system which can assist a programmer in all phases of the programming task in much the same way that a human programmer can. All of the modules described above would be required by such a system, and many more besides. In particular, after requirements, the next problem which needs to be tackled is design. This is needed in order to bridge the gap between requirements and implementation. Once support for requirements and design is available, it should be possible to begin constructing a demonstration system which supports programming as a whole.

Appendices

A - Cliche Library

This appendix presents the complete contents of the Lisp and Ada cliche libraries currently defined as part of KBEmacs. As discussed in Chapter IV, these libraries are much too small to be usable as true prototypes. However, they give a flavor for the kinds of cliches which can be defined.

In the interest of brevity, the discussion below assumes that the reader has read the section of Chapter V which discusses cliche definitions. For the most part, cliches are presented without comment. Only particularly interesting aspects of the cliches are discussed.

Lisp Cliches

Around a third of the cliches in the Lisp cliche library are used in the scenario in Chapter II. These cliches are discussed in detail in that chapter and are only briefly described below. The remaining Lisp cliches are presented here for the first time. The scenario in [Waters 82a] shows many of these latter cliches being used.

The cliches squaring, absolute-value, simple-conditional, and equality-within-epsilon are used as simple examples of cliches in Chapters II & V. They are all straightforward non-looping cliches.

```
(DEFINE-CLICHE SQUARING
    (PRIMARY-ROLES (NUMBER)
     DESCRIBED-ROLES (NUMBER)
     COMMENT "computes the square of {the number}")
  (^ {the input number} 2))

(DEFINE-CLICHE ABSOLUTE-VALUE
    (PRIMARY-ROLES NUMBER
     DESCRIBED-ROLES NUMBER
     COMMENT "computes the absolute value of {number}")
  (LET ((X {input number}))
    (COND ((MINUSP X) (- x)) (T X))))

(DEFINE-CLICHE SIMPLE-CONDITIONAL
    (PRIMARY-ROLES TEST ACTION
     DESCRIBED-ROLES TEST ACTION
     COMMENT "computes {action} if {test} is true")
  (IF {test} {action}))

(DEFINE-CLICHE EQUALITY-WITHIN-EPSILON
    (PRIMARY-ROLES (X Y)
     DESCRIBED-ROLES (X Y)
     COMMENT "determines whether {the x} and {the y}
              differ by less than {the epsilon}"
     CONSTRAINTS ((DEFAULT {the epsilon} 0.00001)))
  (< (ABS (- {the input x} {the input y})) {the epsilon}))
```

The Lisp cliche library contains three additional simple straight-line cliches. The cliche average computes the average of two input numbers.

```
(DEFINE-CLICHE AVERAGE
    (PRIMARY-ROLES (X Y)
     DESCRIBED-ROLES (X Y)
     COMMENT "computes the average of {the x} and {the y}")
  (// (+ {the input x} {the input y}) 2.0))
```

The cliche splice-in splices an element into a list by side-effect. The cliche splice-out performs the reverse operation, splicing an element out of a list by side-effect. Note that both cliches require as an input a pointer to the list cell before the point where an element is to be inserted.

```
(DEFINE-CLICHE SPLICE-IN
    (PRIMARY-ROLES (POSITION NEW-ELEMENT)
     DESCRIBED-ROLES (POSITION NEW-ELEMENT)
     COMMENT "splices {the new-element} into a list after {the position}")
  (LET* ((PREVIOUS-SUBLIST {the input position}))
    (RPLACD PREVIOUS-SUBLIST
            (CONS {the input new-element} (CDR PREVIOUS-SUBLIST)))
    NIL))
```

```
(DEFINE-CLICHE SPLICE-OUT
    (PRIMARY-ROLES (POSITION)
     DESCRIBED-ROLES (POSITION)
     COMMENT "splices an element out of a list after {the position}")
  (LET* ((PREVIOUS-SUBLIST {the input position}))
    (RPLACD PREVIOUS-SUBLIST (CDDR PREVIOUS-SUBLIST))
    NIL))
```

Most of the cliches in the Lisp cliche library are loop fragments. There are three basic kinds of loop fragments *enumerators*, *maps*, and *accumulators*. Enumerators create series of values. Maps compute one series of values from another. Accumulators consume series of values.

The cliche enumeration is a generalized enumerator. It has as empty roles the four roles expected of any enumerator — the input structure, the empty-test, the element-accessor, and the step. (These roles are discussed in the beginning of Chapter II.) The cliche enumeration specifies the way the roles of an enumerator interact without specifying any particular kind of enumeration.

```
(DEFINE-CLICHE ENUMERATION
    (PRIMARY-ROLES (STRUCTURE)
     DESCRIBED-ROLES (STRUCTURE ELEMENT-ACCESSOR EMPTY-TEST STEP)
     COMMENT "enumerates the elements of {the structure}")
  (LET* ((DATA {the input structure}))
    (LOOP DO
      (IF ({the empty-test} DATA) (RETURN))
      {({the element-accessor} DATA), the output element}
      (SETQ DATA ({the step} DATA)))))
```

The next three cliches are examples of particular enumerators. They are each instances of the cliche enumeration. As discussed in the beginning of Chapter II, the cliche list-enumeration enumerates the elements of a list.

```
(DEFINE-CLICHE LIST-ENUMERATION
    (PRIMARY-ROLES (LIST)
     DESCRIBED-ROLES (LIST)
     COMMENT "enumerates the elements of {the list}")
  (LET* ((LIST {the input list}))
    (LOOP DO
      (IF ({NULL, the empty-test} LIST) (RETURN))
      {({CAR, the element-accessor} LIST), the output element}
      (SETQ LIST ({CDR, the step} LIST)))))
```

The cliche sublist-enumeration enumerates the successive sublists of a list. It is almost identical to the cliche list-enumeration. Only the element-accessor is different. In the cliche list-enumeration the element-accessor is the function CAR, whereas in the cliche sublist-enumeration it is the identity function.

```
(DEFINE-CLICHE SUBLIST-ENUMERATION
    (PRIMARY-ROLES (LIST)
     DESCRIBED-ROLES (LIST)
     COMMENT "enumerates the sublists of {the list}")
  (LET* ((LIST {the input list}))
    (LOOP DO
      (IF ({NULL, the empty-test} LIST) (RETURN))
      {{LIST, the element-accessor}, the output sublist}
      (SETQ LIST ({CDR, the step} LIST)))))
```

The cliche vector-enumeration enumerates the elements of a vector. This cliche is interesting in that it requires two variables, VECTOR and I, in order to represent the state of the enumerator.

```
(DEFINE-CLICHE VECTOR-ENUMERATION
    (PRIMARY-ROLES (VECTOR)
     DESCRIBED-ROLES (VECTOR)
     COMMENT "enumerates the elements of {the vector}")
  (LET* ((I 0)
         (VECTOR {the input vector})
         (SIZE (ARRAY-ACTIVE-LENGTH VECTOR)))
    (LOOP DO
      (IF {(NOT (< I SIZE)), the empty-test} (RETURN))
      {({AREF, the element-accessor} VECTOR I), the output element}
      (SETQ I ({1+, the step} I)))))
```

The generalized cliche generation is identical to the cliche enumeration except that it does not have an empty test. It describes a kind of loop fragment, a *generator*, which is a special form of enumerator. A generator generates an unbounded series of values, and is not capable of causing a loop to terminate. Typically, generators are combined with enumerators which control their termination.

```
(DEFINE-CLICHE GENERATION
    (PRIMARY-ROLES (STRUCTURE)
     DESCRIBED-ROLES (STRUCTURE ELEMENT-ACCESSOR STEP)
     COMMENT "generates the elements of {the structure}")
  (LET* ((DATA {the input structure}))
    (LOOP DO
      {({the element-accessor} DATA), the output element}
      (SETQ DATA ({the step} DATA)))))
```

The cliche list-generation is identical to the cliche list-enumeration except that it does not have an empty-test.

```
(DEFINE-CLICHE LIST-GENERATION
    (PRIMARY-ROLES (LIST)
     DESCRIBED-ROLES (LIST)
     COMMENT "generates the elements of {the list}")
  (LET* ((LIST {the input list}))
    (LOOP DO
      {({CAR, the element-accessor} LIST), the output element}
      (SETQ LIST ({CDR, the step} LIST)))))
```

The second basic kind of loop fragment is a map. The basic structure of these fragments is captured in the cliche map. A map applies some function to each element of an input series of values in order to create an output series.

```
(DEFINE-CLICHE MAP
    (PRIMARY-ROLES (ITEM FUNCTION)
     DESCRIBED-ROLES (ITEM FUNCTION)
     COMMENT "maps {the function} over {the item}")
  (LOOP DO
    {({the function} {the input item}), the output result}))
```

The cliche previous-value is an example of specific map fragment. It takes in a series of values and returns a series which contains the same values delayed by one time unit. The cliche takes a special input which specifies what value to use as the first value of the output series. It is interesting to note that the function of the cliche previous-value does not actually perform any computation. It is composed solely of data flow which provides the delay.

```
(DEFINE-CLICHE PREVIOUS-VALUE
    (PRIMARY-ROLES (ITEM ITEM-BEFORE-FIRST-ITEM)
     DESCRIBED-ROLES (ITEM ITEM-BEFORE-FIRST-ITEM)
     COMMENT "keeps track of the previous value of {the item}")
  (LET* ((PREVIOUS {the item-before-first-item}))
    (LOOP DO
      {PREVIOUS, the output previous}
      {(SETQ PREVIOUS {the input item}), the function})))
```

The cliche trailing-pointer-list-enumeration is a specialized enumerator which is a combination of the cliches list-enumeration and previous-value. It is included in the library because it is a relatively common operation when splicing things in and out of lists. An example of this will be shown below.

```
(DEFINE-CLICHE TRAILING-POINTER-LIST-ENUMERATION
    (PRIMARY-ROLES (LIST)
     DESCRIBED-ROLES (LIST)
     COMMENT "enumerates the elements of the tail of {the list}
              maintaining a trailing pointer")
  (LET* ((PREVIOUS-SUBLIST {the input list})
         (LIST (CDR PREVIOUS-SUBLIST)))
    (LOOP DO
      (IF (NULL LIST) (RETURN PREVIOUS-SUBLIST))
      {(CAR LIST), the output element}
      (SETQ PREVIOUS-SUBLIST LIST)
      (SETQ LIST (CDR LIST)))))
```

The cliche group-detector (discussed in Chapter II) is a more complex example of a map. It detects where groups begin in a series of values. Note that the cliche uses the cliche previous-value in order to keep track of the previous value of the input series of values.

```
(DEFINE-CLICHE GROUP-DETECTOR
    (PRIMARY-ROLES (ITEM TEST)
     DESCRIBED-ROLES (ITEM TEST)
     COMMENT "locates the beginning of each group in {the item}")
  (LET* ((DATUM)
         (UNIQUE-VALUE (NCONS NIL))
         (PREVIOUS UNIQUE-VALUE))
    (LOOP DO
      {(PROGN
         (SETQ DATUM {the input item})
         {(OR (EQ PREVIOUS UNIQUE-VALUE) ({the test} PREVIOUS DATUM)),
          the output flag}
         (SETQ PREVIOUS DATUM)), the function})))
```

The cliche selection is a special kind of map fragment. It takes in a series of values and applies a selection-test to each one. It creates an output series which contains only the values for which the selection-test returns non-NIL. The cliche selection is different from the other maps above in that the output series is in general shorter than the input series. An example of the use of the cliche selection will be given below.

```
(DEFINE-CLICHE SELECTION
    (PRIMARY-ROLES (ITEM SELECTION-TEST)
     DESCRIBED-ROLES (ITEM SELECTION-TEST)
     COMMENT "selects a subseries of {the item}")
  (LET* (DATUM)
    (LOOP DO
      (SETQ DATUM {the input item})
      (IF ({the selection-test} DATUM)
          {DATUM, the output selected-item}))))
```

The third and final kind of loop fragment is an accumulator. The basic features of all accumulators are specified in the generalized cliche accumulation. An accumulator takes the values in a series and combines them together into a single value. This is done by using a function (the accumulator) to combine each successive value into an accumulating result.

The result is initialized to the zero of the accumulator function. This yields two nice features. First, if there is only one value in the input series then this value is returned as the accumulated result. Second, if there are no values in the input series then the zero value itself is returned as the result.

```
(DEFINE-CLICHE ACCUMULATION
    (PRIMARY-ROLES (ITEM)
     DESCRIBED-ROLES (ITEM ACCUMULATOR ZERO)
     COMMENT "accumulates {the item}")
  (LET* ((RESULT {the zero}))
    (LOOP DO
      (SETQ RESULT ({the accumulator} RESULT {the input item})))
    RESULT))
```

The cliche sum discussed in Chapter II is an example of a specific accumulator. It adds up the values in a series.

```
(DEFINE-CLICHE SUM
    (PRIMARY-ROLES (NUMBER)
     DESCRIBED-ROLES (NUMBER)
     COMMENT "accumulates the sum of {the number}")
  (LET* ((SUM {0, the zero}))
    (LOOP DO
      (SETQ SUM ({+, the accumulator} SUM {the input number})))
    SUM))
```

The cliche product is very similar to the cliche sum. It multiplies all of the values in a series together. Note that the number 1 is the zero element of the function *.

```
(DEFINE-CLICHE PRODUCT
    (PRIMARY-ROLES (NUMBER)
     DESCRIBED-ROLES (NUMBER)
     COMMENT "accumulates the product of {the number}")
  (LET* ((PRODUCT {1, the zero}))
    (LOOP DO
      (SETQ PRODUCT ({*, the accumulator} PRODUCT {the input number})))
    PRODUCT))
```

The cliche count counts the number of values in a series. The most interesting feature of the cliche count is that, although it depends on the existence of the input values, it does not actually use them in its computation.

```
(DEFINE-CLICHE COUNT
    (PRIMARY-ROLES (ITEM)
     DESCRIBED-ROLES (ITEM)
     COMMENT "accumulates a count of {the item}")
  (LET* ((COUNT {0, the zero}))
    (LOOP DO
      (SETQ COUNT ({+, the accumulator} COUNT {1, depending on the input item})))
    COUNT))
```

As discussed in Chapter V there are a number of features of the plan formalism which are specifically included solely in order to support cliches which depend on values without actually using them. The importance of this can be seen in the following program fragment which is "a count of a selection of a list-enumeration of X".

```
(LET* ((COUNT 0)
       ELEMENT
       (LIST {the input list}))
  (LOOP DO
    (IF (NULL LIST) (RETURN))
    (SETQ ELEMENT (CAR LIST))
    (IF ({the selection-test} ELEMENT) (SETQ COUNT (+ COUNT 1)))
    (SETQ LIST (CDR LIST)))
  COUNT)
```

In the example above, the counting is nested inside of the selection conditional because KBEmacs understands that the count depends on the output of the selection. If it were not for this special knowledge, the counting would be placed outside of the selection conditional because there is no data flow from the selection to the count.

The next cliche, sequential-search, is a common algorithm used in many programs. It is used as an example in Chapter I. Rather than being a loop fragment, the cliche sequential-search specifies an entire loop. The cliche has three roles. The *enumerator* enumerates the elements of a structure. The *search-test* tests each element in order to see if it is the element desired. If an element which passes the search-test is found then an *action* is performed on it.

```
(DEFINE-CLICHE SEQUENTIAL-SEARCH
    (PRIMARY-ROLES (ENUMERATOR)
     DESCRIBED-ROLES (ENUMERATOR SEARCH-TEST ACTION)
     COMMENT "sequentially searches {the input structure}
              looking for an element satisfying {the search-test}")
  (LET* ((DATA {the input structure of the enumerator}))
    (LOOP DO
      (IF ({the empty-test of the enumerator} DATA) (RETURN NIL))
      (IF ({the search-test} DATA) (RETURN {the action}))
      (SETQ DATA ({the step of the enumerator} DATA)))))
```

As an example of using the cliche sequential-search, consider the following set of commands. The commands also use the cliches trailing-pointer-list-enumeration and splice-out. (This set of commands is very similar to an example discussed in [Waters 82a].)

```
Define a sequential-search program DELETE-SYMBOL with a parameter SYMBOL.
Fill the enumerator with
 a trailing-pointer-list-enumeration of (HASH-SYMBOL SYMBOL).
Fill the search-test with (EQ {a use of the element} SYMBOL).
Fill the action with a splice-out of PREVIOUS-SUBLIST.
```

The code which results from the commands is shown below. The program DELETE-SYMBOL deletes a symbol from a hash table by side-effect. This is done by hashing the symbol in order to determine the bucket which contains it and then searching this bucket in order to find an instance of the symbol if any. The cliche trailing-pointer-list enumeration is used so that a pointer to the previous list cell will be available for use by the cliche splice-out.

```
(DEFUN DELETE-SYMBOL (SYMBOL)
  (LET* ((PREVIOUS-SUBLIST (HASH-SYMBOL SYMBOL))
         (LIST (CDR PREVIOUS-SUBLIST)))
    (LOOP DO
      (IF (NULL LIST) (RETURN NIL))
      (WHEN (EQ (CAR LIST) SYMBOL)
        (RPLACD PREVIOUS-SUBLIST (CDDR PREVIOUS-SUBLIST))
        (RETURN NIL))
      (SETQ PREVIOUS-SUBLIST LIST)
      (SETQ LIST (CDR LIST)))))
```

The cliche successive-approximation specifies a particular method of mathematical computation. It uses an *improvement-step* in order to iteratively improve an approximation until a *criterion* is satisfied. The process begins with the choice of an *initial-approximation*.

```
(DEFINE-CLICHE SUCCESSIVE-APPROXIMATION
   (PRIMARY-ROLES (INITIAL-APPROXIMATION)
    DESCRIBED-ROLES (INITIAL-APPROXIMATION CRITERION IMPROVEMENT-STEP)
    COMMENT "iteratively improves an approximation until {the criterion} is
             satisfied")
  (LET* ((APPROXIMATION {the input initial-approximation}))
    (LOOP DO
      (IF ({the criterion} APPROXIMATION) (RETURN APPROXIMATION))
      (SETQ APPROXIMATION ({the improvement-step} APPROXIMATION)))))
```

As an example of using the cliche successive-approximation, consider the following set of commands. The commands also use the cliches average and equality-within-epsilon. (This set of commands is very similar to an example discussed in [Waters 82a].)

```
Define a successive-approximation program SQUARE-ROOT with a parameter NUM.
Fill the initial-approximation with 1.
Fill the improvement-step with
 an average of APPROXIMATION and (// NUM APPROXIMATION).
Fill the criterion with
 an equality-within-epsilon of (* APPROXIMATION APPROXIMATION) and NUM.
```

The code which results from the commands is shown below. The program SQUARE-ROOT computes the square root of a number by using what is essentially a binary search.

```
(DEFUN SQUARE-ROOT (NUM)
  (LET* ((APPROXIMATION 1))
    (LOOP DO
      (IF (< (ABS (- (* APPROXIMATION APPROXIMATION) NUM)) 0.00001)
          (RETURN APPROXIMATION))
      (SETQ APPROXIMATION (// (+ APPROXIMATION (// NUM APPROXIMATION)) 2.0)))))
```

The final three cliches in the Lisp cliche library are the cliches simple-report, print-out, and tabularized-print-out. This suite of cliches is discussed in detail in Chapter II and used to construct the program REPORT-TIMINGS.

```
(DEFINE-CLICHE SIMPLE-REPORT
    (PRIMARY-ROLES (ENUMERATOR PRINT-ITEM SUMMARY)
     DESCRIBED-ROLES (FILE-NAME TITLE ENUMERATOR
                      COLUMN-HEADINGS PRINT-ITEM SUMMARY)
     COMMENT "prints a report of {the input structure of the enumerator}"
     CONSTRAINTS
      ((DEFAULT {the file-name} "report.txt")
       (DERIVED {the line-limit}
                (- 65
                   (SIZE-IN-LINES {the print-item})
                   (SIZE-IN-LINES {the summary})))))
  (WITH-OPEN-FILE (REPORT {the file-name} ':OUT)
    (LET* ((DATE (TIME:PRINT-CURRENT-TIME NIL))
           (LINE 66)
           (PAGE 0)
           (TITLE {the title})
           (DATA {the input structure of the enumerator}))
      (FORMAT REPORT "~5%~66: <~A~>~2%~66: <~A~>~%" TITLE DATE)
      (LOOP DO
        (IF ({the empty-test of the enumerator} DATA) (RETURN))
        (WHEN (> LINE {the line-limit})
          (SETQ PAGE (+ PAGE 1))
          (FORMAT REPORT "~/|~%Page:~3D~50: <~A~>~17A~2%" PAGE TITLE DATE)
          (SETQ LINE 3)
          ({the column-headings} {REPORT, modified} {LINE, modified}))
        ({the print-item} {REPORT, modified}
                          {LINE, modified}
                          ({the element-accessor of the enumerator} DATA))
        (SETQ DATA ({the step of the enumerator} DATA)))
      ({the summary} {REPORT, modified}))))

(DEFINE-CLICHE PRINT-OUT
    (PRIMARY-ROLES (FORMAT-STRING ITEM)
     DESCRIBED-ROLES (FORMAT-STRING ITEM)
     COMMENT "prints out {the item}"
     CONSTRAINTS
      ((DEFAULT {the format-string} "~%~A")
       (DERIVED {the size-in-lines} (SIZE-IN-LINES {the format-string}))))
   (FORMAT REPORT {the format-string} {the input item})
   (SETQ LINE (+ LINE {the size-in-lines})))

(DEFINE-CLICHE TABULARIZED-PRINT-OUT
    (PRIMARY-ROLES (FORMAT-STRING ITEM)
     DESCRIBED-ROLES (FORMAT-STRING ITEM NUMBER-OF-COLUMNS)
     COMMENT "prints out {the item} in columns"
     CONSTRAINTS
      ((DEFAULT {the format-string} "~15A")
       (DERIVED {the maximum-charpos}
                (- 75 (SIZE-IN-CHARACTERS {the format-string})))))
  (WHEN (> (CHARPOS REPORT) {the maximum-charpos})
    (FORMAT REPORT "~&")
    (SETQ LINE (+ LINE 1)))
  (FORMAT REPORT {the format-string} {the input item}))
```

Ada Cliches

Except for the cliche equality_within_epsilon which is used as an example in Chapter V, all of the cliches in the Ada cliche library are used in the scenario in Chapter III and, for the most part, are discussed in detail there.

The cliche equality_within_epsilon is identical to the cliche equality-within-epsilon except for the fact that it is rendered in Ada syntax.

```
cliche EQUALITY_WITHIN_EPSILON is
   primary roles X, Y;
   described roles X, Y, EPSILON;
   comment "determines whether {the x} and {the y}
            differ by less than {the epsilon}";
   constraints
      DEFAULT({the epsilon}, 0.00001);
   end constraints;

begin
   return abs({the input x} - {the input y}) < {the epsilon};
end EQUALITY_WITHIN_EPSILON;
```

The cliche read reads a record from a file. Like all of the Ada cliches which access files, the cliche read depends on the existence of a set of conventions (described in the beginning of Chapter III) governing file I/O. The cliche is surprisingly complex due both to the verbose nature of Ada I/O procedures and the need to properly handle interrupts which can occur during I/O.

```
cliche READ is
   primary roles FILE, KEY;
   described roles FILE, KEY;
   comment "reads the record indexed by {the key} from {the file}";
   constraints
      RENAME("DATA_RECORD", SINGULAR_FORM({the file}));
      DEFAULT({the file_name}, CORRESPONDING_FILE_NAME({the file}));
   end constraints;

   DATA_RECORD: {};
   FILE: {};
begin
   FILE := {the file};
   OPEN(FILE, IN_FILE, {the file_name});
   READ(FILE, DATA_RECORD, {the key});
   {DATA_RECORD, the output data_record};
   CLOSE(FILE);
exception
   when DEVICE_ERROR | END_ERROR | NAME_ERROR | STATUS_ERROR =>
      CLOSE(FILE); PUT("Data Base Inconsistent");
   when others => CLOSE(FILE); raise;
end READ;
```

As discussed in Chapter III, the cliche pre_loop_computation exists in order to allow a programmer to conveniently specify an initialization computation for a loop inside of an expression nested inside the loop.

```
cliche PRE_LOOP_COMPUTATION is
   primary roles ITEM;
   described roles ITEM;
   comment "makes the value {the item} available inside of a loop";
   constraints
      RENAME("DATA_VALUE", SUGGEST_VARIABLE_NAME({the item}));
   end constraints;

   DATA_VALUE: {};
begin
   DATA_VALUE := {the item};
   loop
      {DATA_VALUE, the output data_value};
   end loop;
end PRE_LOOP_COMPUTATION;
```

The cliche query_user_for_key asks the user of a program to supply the key for a record. In addition, it checks the key in order to ensure that it is a valid key. This cliche is used both in the program UNIT_REPAIR_REPORT and the program MODEL_DEFECTS_REPORT in Chapter III.

```
with TEXT_IO;
use TEXT_IO;
cliche QUERY_USER_FOR_KEY is
   primary roles FILE;
   described roles FILE;
   comment "queries the user for a key to a record in {the file}";
   constraints
      RENAME("DATA_RECORD", SINGULAR_FORM({the file}));
      DEFAULT({the file_name}, CORRESPONDING_FILE_NAME({the file}));
   end constraints;

   DATA_RECORD: {};
   DATA_RECORD_KEY: {};
   FILE: {};
begin
   FILE := {the file};
   OPEN(FILE, IN_FILE, {the file_name});
   loop
      begin
         NEW_LINE; PUT("Enter DATA_RECORD Key: "); GET(DATA_RECORD_KEY);
         READ(FILE, DATA_RECORD, DATA_RECORD_KEY);
         exit;
      exception
         when END_ERROR => PUT("Invalid DATA_RECORD Key"); NEW_LINE;
      end;
   end loop;
   CLOSE(FILE);
   return DATA_RECORD_KEY;
exception
   when DEVICE_ERROR | END_ERROR | NAME_ERROR | STATUS_ERROR =>
      CLOSE(FILE); PUT("Data Base Inconsistent");
   when others => CLOSE(FILE); raise;
end QUERY_USER_FOR_KEY;
```

The cliches chain_file_definition and keyed_file_definition are both data cliches which specify how to define a file of records. The cliche chain_file_definition is discussed in detail in Chapter III. The cliche keyed_file_definition is very similar. Both cliches rely heavily on generic packages defined in the package FUNCTIONS (see Appendix B).

```
with FUNCTIONS;
use FUNCTIONS;
cliche CHAIN_FILE_DEFINITION is
   primary roles FILE_NAME;
   described roles FILE_NAME;
   comment "defines a file named {the file_name} of chain records";
   constraints
      RENAME("DATA_RECORD", FILE_NAME_ROOT({the file_name}));
   end constraints;

   DATA_RECORDS_NAME: constant STRING := {the file_name};
   subtype DATA_RECORD_INDEX_TYPE is INDEX_TYPE;
   type DATA_RECORD_TYPE is
      record
         {the data};
         NEXT: DATA_RECORD_INDEX_TYPE;
      end record;
   package DATA_RECORD_IO is
      new CHAINED_IO(DATA_RECORD_TYPE, DATA_RECORD_INDEX_TYPE);
   DATA_RECORDS: DATA_RECORD_IO.FILE_TYPE;
end CHAIN_FILE_DEFINITION;

with FUNCTIONS;
use FUNCTIONS;
cliche KEYED_FILE_DEFINITION is
   primary roles FILE_NAME;
   described roles FILE_NAME;
   comment "defines a keyed file named {the file_name} of records";
   constraints
      RENAME("DATA_RECORD", FILE_NAME_ROOT({the file_name}));
   end constraints;

   DATA_RECORDS_NAME: constant STRING := {the file_name};
   subtype DATA_RECORD_KEY_TYPE is STRING(1..{the key_length});
   type DATA_RECORD_TYPE is
      record
         {the data};
      end record;
   package DATA_RECORD_IO is
      new KEYED_IO(DATA_RECORD_TYPE, DATA_RECORD_KEY_TYPE);
   DATA_RECORDS: DATA_RECORD_IO.FILE_TYPE;
end KEYED_FILE_DEFINITION;
```

The cliche file_enumeration is an enumerator which reads a file sequentially, enumerating the records in it. As discussed in Chapter III it has the same basic structure as any other enumerator.

```
cliche FILE_ENUMERATION is
   primary roles FILE;
   described roles FILE;
   comment "enumerates the records in {the file}";
   constraints
      RENAME("DATA_RECORD", SINGULAR_FORM({the file}));
      DEFAULT({the file_name}, CORRESPONDING_FILE_NAME({the file}));
   end constraints;

   FILE: {};
   DATA_RECORD: {};
begin
   FILE := {the input file};
   OPEN(FILE, IN_FILE, {the file_name});
   while not {END_OF_FILE, the empty_test}(FILE) loop
      {{READ, the element_accessor}, the step}(FILE, DATA_RECORD);
      {DATA_RECORD, the output data_record};
   end loop;
   CLOSE(FILE);
exception
   when DEVICE_ERROR | END_ERROR | NAME_ERROR | STATUS_ERROR =>
      CLOSE(FILE); PUT("Data Base Inconsistent");
   when others => CLOSE(FILE); raise;
end FILE_ENUMERATION;
```

The cliche file_accumulation is an accumulator. It is very similar the the cliche file_enumeration except that it essentially performs the inverse operation. It takes in a series of records and writes them sequentially into a file.

```
with DIRECT_IO;
cliche FILE_ACCUMULATION is
   primary roles FILE;
   described roles FILE, DATA_RECORD;
   comment "creates a file containing {the data_record}";
   constraints
      RENAME("DATA_RECORD", SINGULAR_FORM({the file}));
      DEFAULT({the file_name}, CORRESPONDING_FILE_NAME({the file}));
   end constraints;

   FILE: {};
   DATA_RECORD: {};
begin
   FILE := {the file};
   {OPEN, the zero}(FILE, OUT_FILE, {the file_name});
   loop
      {WRITE, the accumulator}(FILE, {the data_record});
   end loop;
   {FILE, the output resulting_file};
   CLOSE(FILE);
exception
   when DEVICE_ERROR | END_ERROR | NAME_ERROR | STATUS_ERROR =>
      CLOSE(FILE); PUT("Data Base Inconsistent");
   when others => CLOSE(FILE); raise;
end FILE_ACCUMULATION;
```

The cliche chain_enumeration (which is discussed in detail in the beginning of Chapter III) enumerates the records in a chain. An interesting aspect of the cliche is that it makes use of a number of constraints in order to fill in roles of the cliche based on the definitions for the files referenced.

```
cliche CHAIN_ENUMERATION is
   primary roles MAIN_FILE, CHAIN_FILE, MAIN_FILE_KEY;
   described roles MAIN_FILE, CHAIN_FILE, MAIN_FILE_KEY;
   comment "enumerates the chain records in {the chain_file} starting
            from the header record indexed by {the main_file_key}";
   constraints
      RENAME("MAIN_RECORD", SINGULAR_FORM({the main_file}));
      RENAME("CHAIN_RECORD", SINGULAR_FORM({the chain_file}));
      DEFAULT({the main_file_name},
              CORRESPONDING_FILE_NAME({the main_file}));
      DEFAULT({the chain_file_name},
              CORRESPONDING_FILE_NAME({the chain_file}));
      DEFAULT({the main_file_chain_field},
              CHAIN_FIELD({the main_file}, {the chain_file}));
      DEFAULT({the step}, CHAIN_FIELD({the chain_file}, {the chain_file}));
   end constraints;

   CHAIN_FILE: {};
   CHAIN_RECORD: {};
   CHAIN_RECORD_INDEX: {};
   MAIN_FILE: {};
   MAIN_RECORD: {};

   procedure CLEAN_UP is
   begin
      CLOSE(CHAIN_FILE); CLOSE(MAIN_FILE);
   exception
      when STATUS_ERROR => return;
   end CLEAN_UP;
begin
   CHAIN_FILE := {the chain_file};
   MAIN_FILE := {the main_file};
   OPEN(CHAIN_FILE, IN_FILE, {the chain_file_name});
   OPEN(MAIN_FILE, IN_FILE, {the main_file_name});
   READ(MAIN_FILE, MAIN_RECORD, {the input main_file_key});
   CHAIN_RECORD_INDEX := MAIN_RECORD.{the main_file_chain_field};
   while not {NULL_INDEX, the empty_test}(CHAIN_RECORD_INDEX) loop
      {READ, the element_accessor}(CHAIN_FILE, CHAIN_RECORD, CHAIN_RECORD_INDEX);
      {CHAIN_RECORD, the output chain_record};
      CHAIN_RECORD_INDEX := CHAIN_RECORD.{the step};
   end loop;
   CLEAN_UP;
exception
   when DEVICE_ERROR | END_ERROR | NAME_ERROR | STATUS_ERROR =>
      CLEAN_UP; PUT("Data Base Inconsistent");
   when others => CLEAN_UP; raise;
end CHAIN_ENUMERATION;
```

The cliche all_chains_enumeration is similar to the cliche chain_enumeration except that it enumerates all of the records in a chain file. It contains a carefully crafted inner loop which creates a one dimensional series of the chain records based on the inherently two dimensional structure of chains hanging off of main file records.

```
with FUNCTIONS;
use FUNCTIONS;
cliche ALL_CHAINS_ENUMERATION is
   primary roles MAIN_FILE, CHAIN_FILE;
   described roles MAIN_FILE, CHAIN_FILE;
   comment "enumerates the header records in {the main_file} and
            enumerates the chain records in {the chain_file} starting
            from each header record";
   constraints
      RENAME("MAIN_RECORD", SINGULAR_FORM({the main_file}));
      RENAME("CHAIN_RECORD", SINGULAR_FORM({the chain_file}));
      DEFAULT({the main_file_name},
              CORRESPONDING_FILE_NAME({the main_file}));
      DEFAULT({the chain_file_name},
              CORRESPONDING_FILE_NAME({the chain_file}));
      DEFAULT({the main_file_chain_field},
              CHAIN_FIELD({the main_file}, {the chain_file}));
      DEFAULT({the chain_file_chain_field},
              CHAIN_FIELD({the chain_file}, {the chain_file}));
   end constraints;

   CHAIN_FILE: {};
   CHAIN_RECORD: {};
   CHAIN_RECORD_INDEX: {};
   MAIN_FILE: {};
   MAIN_RECORD: {};

   procedure CLEAN_UP is
   begin
      CLOSE(CHAIN_FILE); CLOSE(MAIN_FILE);
   exception
      when STATUS_ERROR => return;
   end CLEAN_UP;
begin
   CHAIN_FILE := {the chain_file};
   MAIN_FILE := {the main_file};
   OPEN(CHAIN_FILE, IN_FILE, {the chain_file_name});
   OPEN(MAIN_FILE, IN_FILE, {the main_file_name});
   CHAIN_RECORD_INDEX := NULL_INDEX;
   loop
      while NULL_INDEX(CHAIN_RECORD_INDEX) and not END_OF_FILE(MAIN_FILE) loop
         READ(MAIN_FILE, MAIN_RECORD);
         CHAIN_RECORD_INDEX := MAIN_RECORD.{the main_file_chain_field};
      end loop;
      exit when NULL_INDEX(CHAIN_RECORD_INDEX);
      READ(CHAIN_FILE, CHAIN_RECORD, CHAIN_RECORD_INDEX);
      {CHAIN_RECORD, the output chain_record};
      CHAIN_RECORD_INDEX := CHAIN_RECORD.{the chain_file_chain_field};
   end loop;
   CLEAN_UP;
exception
   when DEVICE_ERROR | END_ERROR | NAME_ERROR | STATUS_ERROR =>
      CLEAN_UP; PUT("Data Base Inconsistent");
   when others => CLEAN_UP; raise;
end ALL_CHAINS_ENUMERATION;
```

As discussed at length in Chapter III, the cliche simple_report is very similar to the cliche simple-report.

```
with CALENDAR, FUNCTIONS, TEXT_IO;
use CALENDAR, FUNCTIONS, TEXT_IO;
cliche SIMPLE_REPORT is
   primary roles ENUMERATOR, PRINT_ITEM, SUMMARY;
   described roles FILE_NAME, TITLE, ENUMERATOR, COLUMN_HEADINGS,
                   PRINT_ITEM, SUMMARY;
   comment "prints a report of {the input structure of the enumerator}";
   constraints
      DEFAULT({the file_name}, "report.txt");
      DERIVED({the line_limit},
              66-SIZE_IN_LINES({the print_item})
                -SIZE_IN_LINES({the summary}));
      DEFAULT({the print_item}, CORRESPONDING_PRINTING({the enumerator}));
      DEFAULT({the column_headings},
              CORRESPONDING_HEADINGS({the print_item}));
   end constraints;

   use INT_IO;
   CURRENT_DATE: constant STRING := FORMAT_DATE(CLOCK);
   DATA: {};
   REPORT: TEXT_IO.FILE_TYPE;
   TITLE: STRING(1..{});
   procedure CLEAN_UP is
   begin
      SET_OUTPUT(STANDARD_OUTPUT);
      CLOSE(REPORT);
   exception
      when STATUS_ERROR => return;
   end CLEAN_UP;
begin
   CREATE(REPORT, OUT_FILE, {the file_name});
   DATA := {the input structure of the enumerator};
   SET_OUTPUT(REPORT);
   TITLE := {the title};
   NEW_LINE(4); SET_COL(20); PUT(CURRENT_DATE); NEW_LINE(2);
   SET_COL(13); PUT(TITLE); NEW_LINE(60);
   while not {the empty_test of the enumerator}(DATA) loop
      if LINE > {the line_limit} then
         NEW_PAGE; NEW_LINE; PUT("Page: "); PUT(INTEGER(PAGE-1), 3);
         SET_COL(13); PUT(TITLE);
         SET_COL(61); PUT(CURRENT_DATE); NEW_LINE(2);
         {the column_headings}({CURRENT_OUTPUT, modified});
      end if;
      {the print_item}({CURRENT_OUTPUT, modified},
                       {the element_accessor of the enumerator}(DATA));
      DATA := {the step of the enumerator}(DATA);
   end loop;
   {the summary};
   CLEAN_UP;
exception
   when DEVICE_ERROR | END_ERROR | NAME_ERROR | STATUS_ERROR =>
      CLEAN_UP; PUT("Data Base Inconsistent");
   when others => CLEAN_UP; raise;
end SIMPLE_REPORT;
```

The next two cliches (file_selection and file_summarization) are specialized cliches used during the construction of the program MODEL_DEFECTS_REPORT in Chapter III. They are discussed in detail in that chapter.

```
with DIRECT_IO;
cliche FILE_SELECTION is
   primary roles SOURCE_FILE;
   described roles SOURCE_FILE, SELECTION_TEST, RECORD_KEY_ACCESSOR;
   comment "creates a file containing selected record keys for
            {the source_file}";
   constraints
      RENAME("RECORD_KEY", RECORD_KEY_ROOT({the source_file}));
   end constraints;

   package SELECTION_IO is new DIRECT_IO(RECORD_KEY_TYPE);
   use SELECTION_IO;
   SELECTIONS: SELECTION_IO.FILE_TYPE;

   DATA: {};
   DATUM: {};
begin
   DATA := {the source_file};
   CREATE(SELECTIONS, INOUT_FILE);
   while not {the empty_test of the enumerator}(DATA) loop
      DATUM := {the element_accessor of the enumerator}(DATA);
      if {the selection_test}(DATUM) then
         WRITE(SELECTIONS, {the record_key_accessor}(DATUM));
      end if;
      DATA := {the step of the enumerator}(DATA);
   end loop;
   {SELECTIONS, the output selection_file};
   CLOSE(SELECTIONS);
exception
   when DEVICE_ERROR | END_ERROR | NAME_ERROR | STATUS_ERROR =>
      CLOSE(SELECTIONS); PUT("Data Base Inconsistent");
   when others => CLOSE(SELECTIONS); raise;
end FILE_SELECTION;
```

```
with DIRECT_IO;
cliche FILE_SUMMARIZATION is
   primary roles SOURCE_FILE;
   described roles SOURCE_FILE;
   comment "creates a file summarizing the frequency of occurrence of
            the record keys stored in {the source file}";
   constraints
      RENAME("SOURCE_RECORD", SINGULAR_FORM({the source_file}));
      RENAME("RECORD_KEY", RECORD_KEY_ROOT({the source_file}));
      DEFAULT({the source_file_name},
              CORRESPONDING_FILE_NAME({the source_file}));
   end constraints;

   type SUMMARY_TYPE is
      record
         COUNT: INTEGER;
         KEY: RECORD_KEY_TYPE;
      end record;
   package SUMMARY_IO is new DIRECT_IO(SUMMARY_TYPE);
   use SUMMARY_IO;
   SUMMARIES: SUMMARY_IO.FILE_TYPE;

   SOURCE_FILE: {};

   function BUILD_SUMMARY(COUNT: INTEGER; KEY: RECORD_KEY_TYPE)
                             return SUMMARY_TYPE is
      begin return SUMMARY_TYPE'(COUNT, KEY); end BUILD_SUMMARY;
   function SUMMARY_GREATER_THAN(X: SUMMARY_TYPE; Y: SUMMARY_TYPE)
                                    return BOOLEAN is
      begin return X.COUNT > Y.COUNT; end SUMMARY_GREATER_THAN;
   procedure SORT_SOURCE_RECORDS is
      new SORT_FILE(RECORD_KEY_TYPE, SOURCE_RECORD_IO.FILE_TYPE,
                    SOURCE_RECORD_IO.POSITIVE_COUNT);
   procedure SORT_SUMMARIES is
      new SORT_FILE(SUMMARY_TYPE, SUMMARY_IO.FILE_TYPE,
                    SUMMARY_IO.POSITIVE_COUNT, SUMMARY_GREATER_THAN);
   procedure SUMMARIZE_SOURCE_RECORDS is
      new SUMMARIZE_FILE(RECORD_KEY_TYPE, SOURCE_RECORD_IO.FILE_TYPE,
                         SUMMARY_TYPE, SUMMARY_IO.FILE_TYPE, BUILD_SUMMARY);
   procedure CLEAN_UP is
      begin
         CLOSE(SOURCE_FILE); CLOSE(SUMMARIES);
      exception
         when STATUS_ERROR => return;
      end CLEAN_UP;
begin
   SOURCE_FILE := {the source_file};
   OPEN(SOURCE_FILE, INOUT_FILE, {the source_file_name});
   CREATE(SUMMARIES, INOUT_FILE);
   SORT_SOURCE_RECORDS(SOURCE_FILE);
   SUMMARIZE_SOURCE_RECORDS(SOURCE_FILE, SUMMARIES);
   SORT_SUMMARIES(SUMMARIES);
   {SUMMARIES, the output summaries};
   CLEAN_UP;
exception
   when DEVICE_ERROR | END_ERROR | NAME_ERROR | STATUS_ERROR =>
      CLEAN_UP; PUT("Data Base Inconsistent");
   when others => CLEAN_UP; raise;
end FILE_SUMMARIZATION;
```

B - Supporting Functions

This appendix shows the supporting functions which are required in order to make the example programs in the scenarios in Chapters II & III run. Particularly for Ada, understanding these functions is a prerequisite for gaining a complete understanding of the programs in the scenarios.

Lisp Functions

The Lisp programs in Chapter II are built almost entirely out of standard Lisp Machine functions. Only two non-standard functions are used.

The function DISPLAY-REPORT reads in the file "report.txt" and prints it out on the screen. It is useful for debugging report programs. The only complicated thing about the program is that it abbreviates groups of consecutive blank lines.

```
(DEFUN DISPLAY-REPORT ()
  (WITH-OPEN-FILE (FILE "report.txt" ':IN)
    (DO ((CHAR (TYI FILE NIL) (TYI FILE NIL))
         (NEWLINES 1))
        (NIL)
      (DO () ((NOT (EQ CHAR #EWLINE)))
        (INCF NEWLINES)
        (SETQ CHAR (TYI FILE NIL)))
      (COND ((= NEWLINES 1) (TERPRI))
            ((= NEWLINES 2) (TERPRI) (TERPRI))
            ((> NEWLINES 2)
             (FORMAT T "~%<~D blank lines>~%" (1- NEWLINES))))
      (SETQ NEWLINES 0)
      (IF (NULL CHAR) (RETURN))
      (TYO CHAR)))
  (VALUES))
```

The function CHARPOS queries an output file in order to determine the line position where the next character will go. It can be looked at as merely an abbreviation for the standard Lisp Machine message shown. (The name of this function is carried over from MacLisp.)

```
(DEFUN CHARPOS (STREAM)
  (SEND STREAM ':READ-CURSORPOS ':CHARACTER))
```

Ada Functions

Several non-standard Ada functions and packages are used in the scenario in Chapter III. The procedure CREATE_TEST_RECORDS creates a set of test of records (shown in Figure 7) in the files DEFECTS, MODELS, REPAIRS, and UNITS. These records are used to test the programs UNIT_REPAIR_REPORT and MODEL_DEFECTS_REPORT in the scenario in Chapter III.

```
with CALENDAR, MAINTENANCE_FILES, TEXT_IO;
use CALENDAR, MAINTENANCE_FILES, TEXT_IO;
procedure CREATE_TEST_RECORDS is
   use DEFECT_IO, MODEL_IO, REPAIR_IO, UNIT_IO;

   procedure CLEAN_UP is
      begin
         CLOSE(DEFECTS); CLOSE(MODELS); CLOSE(REPAIRS); CLOSE(UNITS);
      exception
         when STATUS_ERROR => return;
      end CLEAN_UP;
begin
   CREATE(DEFECTS, OUT_FILE, DEFECTS_NAME);
   CREATE(MODELS, OUT_FILE, MODELS_NAME);
   CREATE(REPAIRS, OUT_FILE, REPAIRS_NAME);
   CREATE(UNITS, OUT_FILE, UNITS_NAME);
   WRITE(MODELS, MODEL_TYPE'("Opal Sorter     ", "Perth Mining    "), "OS1");
   WRITE(MODELS, MODEL_TYPE'("Gas Analyzer    ", "Benson Labs     "), "GA2");
   WRITE(DEFECTS, DEFECT_TYPE'("Power supply thermistor blown    ", "OS1"),
         "OS-03");
   WRITE(DEFECTS, DEFECT_TYPE'("Control board cold solder joint ", "GA2"),
         "GA-11");
   WRITE(DEFECTS, DEFECT_TYPE'("Clogged gas injection port      ", "GA2"),
         "GA-32");
   WRITE(REPAIRS,
         REPAIR_TYPE'(TIME_OF(1983, 9, 14), "GA-32",
                      "Probably caused by humidity.    ", 0),
         1);
   WRITE(REPAIRS,
         REPAIR_TYPE'(TIME_OF(1985, 1, 23), "GA-11",
                      "Took two days to find.          ", 1),
         2);
   WRITE(REPAIRS,
         REPAIR_TYPE'(TIME_OF(1985, 2, 25), "OS-03",
                      "Sorter arm got stuck.           ", 0),
         3);
   WRITE(REPAIRS,
         REPAIR_TYPE'(TIME_OF(1985, 3, 19), "GA-32",
                      "Port Diameter seems below specs.", 2),
         4);
   WRITE(UNITS, UNIT_TYPE'("OS1", 3), "OS1-271");
   WRITE(UNITS, UNIT_TYPE'("GA2", 4), "GA2-342");
   CLEAN_UP;
exception
   when DEVICE_ERROR | END_ERROR | NAME_ERROR | STATUS_ERROR =>
      CLEAN_UP; PUT("Data Base Inconsistent");
   when others => CLEAN_UP; raise;
end CREATE_TEST_RECORDS;
```

The procedure DISPLAY_REPORT is essentially identical to the function DISPLAY-REPORT except that it is rendered in Ada syntax.

```
with TEXT_IO;
use TEXT_IO;
procedure DISPLAY_REPORT is
   package INT_IO is new INTEGER_IO(INTEGER);
   use INT_IO;
   CHAR: CHARACTER;
   LINE_TERMINATORS: COUNT;
   REPORT: TEXT_IO.FILE_TYPE;
begin
   OPEN(REPORT, IN_FILE, "report.txt");
   LOOP
      LINE_TERMINATORS := 0;
      while not END_OF_FILE(REPORT) and then END_OF_LINE(REPORT) loop
         SKIP_LINE(REPORT);
         LINE_TERMINATORS := LINE_TERMINATORS+1;
      end loop;
      if LINE_TERMINATORS > 2 then
         NEW_LINE; PUT("<"); PUT(INTEGER(LINE_TERMINATORS-1), 2);
         PUT(" blank lines>"); NEW_LINE;
      elsif LINE_TERMINATORS > 0 then
         NEW_LINE(LINE_TERMINATORS);
      end if;
      exit when END_OF_FILE(REPORT);
      GET(REPORT, CHAR);
      PUT(CHAR);
   end loop;
   CLOSE(REPORT);
end DISPLAY_REPORT;
```

The package FUNCTIONS is an integral part of the scenario in Chapter III. It defines a number of non-standard functions and packages which are used in the programs created. The package declaration for the package FUNCTIONS is shown on the next two pages.

In order to support I/O for integers, the package FUNCTIONS creates a package INT_IO which is an instantiation of the standard Ada package INTEGER_IO. The package FUNCTIONS then defines a type INDEX_TYPE which is used when instantiating chain files. (See the definition of the package MAINTENANCE_FILES in Chapter III.)

The function FORMAT_DATE converts times into strings of the form "mm/dd/yyyy". The type DATE_STRING is used to declare variables containing strings of this form.

The function QUERY_USER_FOR_DATE prints out a prompt string and reads in a date. This operation is represented as a function instead of as a cliche for two reasons. First, in contrast to query_user_for_key, QUERY_USER_FOR_DATE does not require any file I/O and therefore can be efficiently realized as a subroutine. Second, QUERY_USER_FOR_DATE is a very simple operation which is unlikely to be modified by a programmer. As a result, there is no advantage to having it expand in-line.

The generic procedures SORT_FILE and SUMMARIZE_FILE are used by the cliche FILE_SUMMARIZATION. These operations are represented as generic procedures rather than as cliches because the generic mechanism is fully adequate for representing them. The use of cliches is reserved for situations where other Ada mechanisms are not sufficient.

The generic procedure SORT_FILE sorts the records in a file. It takes a functional parameter (">") which specifies how to order the records. The generic procedure SUMMARIZE_FILE creates a summary file. (The operation of summarization is discussed in Chapter III.)

The generic packages CHAINED_IO and KEYED_IO define the operations which are required to operate on chain file and keyed files respectively. These packages are instantiated when defining the files in the package MAINTENANCE_FILES.

```
with CALENDAR, IO_EXCEPTIONS, TEXT_IO;
use CALENDAR, TEXT_IO;
package FUNCTIONS is
   package INT_IO is new INTEGER_IO(INTEGER);
   subtype INDEX_TYPE is INTEGER range 1..INTEGER'LAST;
   subtype DATE_STRING_TYPE is STRING(1..10);
   function FORMAT_DATE(DATE: TIME) return DATE_STRING_TYPE;
   function QUERY_USER_FOR_DATE(PROMPT: STRING) return TIME;

   generic
      type ITEM_TYPE is private;
      type FILE_TYPE is limited private;
      type INDEX_TYPE is range <>;
      with function ">"(X: ITEM_TYPE; Y: ITEM_TYPE) return BOOLEAN is <>;
      with procedure READ(FILE: FILE_TYPE; ITEM: out ITEM_TYPE) is <>;
      with procedure WRITE(FILE: FILE_TYPE; ITEM: ITEM_TYPE;
                           TO: INDEX_TYPE) is <>;
      with procedure SET_INDEX(FILE: FILE_TYPE; TO: INDEX_TYPE) is <>;
      with function INDEX(FILE: FILE_TYPE) RETURN INDEX_TYPE is <>;
      with function END_OF_FILE(FILE: FILE_TYPE) return BOOLEAN is <>;
   procedure SORT_FILE(FILE: FILE_TYPE);

   generic
      type FROM_ITEM_TYPE is private;
      type FROM_FILE_TYPE is limited private;
      type TO_ITEM_TYPE is limited private;
      type TO_FILE_TYPE is limited private;
      with function BUILD(COUNT: INTEGER; ITEM: FROM_ITEM_TYPE)
                          return TO_ITEM_TYPE;
      with procedure READ(FILE: FROM_FILE_TYPE; ITEM: out FROM_ITEM_TYPE) is <>;
      with procedure WRITE(FILE: TO_FILE_TYPE; ITEM: TO_ITEM_TYPE) is <>;
      with function END_OF_FILE(FILE: FROM_FILE_TYPE) return BOOLEAN is <>;
      with procedure RESET(FILE: TO_FILE_TYPE) is <>;
   procedure SUMMARIZE_FILE(FROM_FILE: FROM_FILE_TYPE; TO_FILE: TO_FILE_TYPE);
```

```
   generic
      type ELEMENT_TYPE is private;
      type INDEX_TYPE is range <>;
   package CHAINED_IO is
      package UNDERLYING_IO is new DIRECT_IO(ELEMENT_TYPE);
      type FILE_TYPE is new UNDERLYING_IO.FILE_TYPE;
      type FILE_MODE is new UNDERLYING_IO.FILE_MODE;

      procedure CREATE(FILE: in out FILE_TYPE; MODE: FILE_MODE; NAME: STRING);
      procedure OPEN(FILE: in out FILE_TYPE; MODE: FILE_MODE; NAME: STRING);
      procedure CLOSE(FILE: in out FILE_TYPE);
      procedure READ(FILE: FILE_TYPE; ITEM: out ELEMENT_TYPE; FROM: INDEX_TYPE);
      procedure WRITE(FILE: FILE_TYPE; ITEM: ELEMENT_TYPE; TO: INDEX_TYPE);
      function END_OF_FILE(FILE: FILE_TYPE) return BOOLEAN;
      function FREE_INDEX(FILE: FILE_TYPE) return INDEX_TYPE;
      function NULL_INDEX(INDEX: INDEX_TYPE) return BOOLEAN;
      function NULL_INDEX return INDEX_TYPE;

      STATUS_ERROR: exception renames IO_EXCEPTIONS.STATUS_ERROR;
      MODE_ERROR: exception renames IO_EXCEPTIONS.MODE_ERROR;
      NAME_ERROR: exception renames IO_EXCEPTIONS.NAME_ERROR;
      USE_ERROR: exception renames IO_EXCEPTIONS.USE_ERROR;
      DEVICE_ERROR: exception renames IO_EXCEPTIONS.DEVICE_ERROR;
      END_ERROR: exception renames IO_EXCEPTIONS.END_ERROR;
      DATA_ERROR: exception renames IO_EXCEPTIONS.DATA_ERROR;
   end CHAINED_IO;

   generic
      type ELEMENT_TYPE is private;
      type KEY_TYPE is private;
   package KEYED_IO is
      type COMBINED_TYPE is
         record
            KEY: KEY_TYPE;
            ELEMENT: ELEMENT_TYPE;
         end record;
      package UNDERLYING_IO is new DIRECT_IO(COMBINED_TYPE);
      type FILE_TYPE is new UNDERLYING_IO.FILE_TYPE;
      type FILE_MODE is new UNDERLYING_IO.FILE_MODE;

      procedure CREATE(FILE: in out FILE_TYPE; MODE: FILE_MODE; NAME: STRING);
      procedure OPEN(FILE: in out FILE_TYPE; MODE: FILE_MODE; NAME: STRING);
      procedure CLOSE(FILE: in out FILE_TYPE);
      procedure READ(FILE: FILE_TYPE; ITEM: out ELEMENT_TYPE; FROM: KEY_TYPE);
      procedure READ(FILE: FILE_TYPE; ITEM: out ELEMENT_TYPE);
      procedure WRITE(FILE: FILE_TYPE; ITEM: ELEMENT_TYPE; TO: KEY_TYPE);
      procedure SET_KEY(FILE: FILE_TYPE; KEY: KEY_TYPE);
      function END_OF_FILE(FILE: FILE_TYPE) return BOOLEAN;

      STATUS_ERROR: exception renames IO_EXCEPTIONS.STATUS_ERROR;
      MODE_ERROR: exception renames IO_EXCEPTIONS.MODE_ERROR;
      NAME_ERROR: exception renames IO_EXCEPTIONS.NAME_ERROR;
      USE_ERROR: exception renames IO_EXCEPTIONS.USE_ERROR;
      DEVICE_ERROR: exception renames IO_EXCEPTIONS.DEVICE_ERROR;
      END_ERROR: exception renames IO_EXCEPTIONS.END_ERROR;
      DATA_ERROR: exception renames IO_EXCEPTIONS.DATA_ERROR;
   end KEYED_IO;
end FUNCTIONS;
```

The next four pages show the body of the package FUNCTIONS. The functions FORMAT_DATE and QUERY_USER_FOR_DATE are straightforward. The generic function SORT_FILE uses an extremely simple but inefficient selection sort. The generic function summarize_file is based on a modified form of the cliche group detector. It counts up the number of elements in each group. The generic packages CHAINED_IO and KEYED_IO are large because they have to define all of the I/O functions to be used on these types of files. However, each of the I/O functions is by itself trivial. All of the functions in the package FUNCTIONS are intended merely to be minimal support functions, and not examples of good style.

```
package body FUNCTIONS is
   function FORMAT_DATE(DATE: TIME) return DATE_STRING_TYPE is
         use INT_IO;
         MONTH_STRING: STRING(1..2);
         DAY_STRING: STRING(1..2);
         YEAR_STRING: STRING(1..4);
      begin
         PUT(MONTH_STRING, MONTH(DATE)); PUT(DAY_STRING, DAY(DATE));
         PUT(YEAR_STRING, YEAR(DATE));
         return MONTH_STRING & "/" & DAY_STRING & "/" & YEAR_STRING;
      end FORMAT_DATE;

   function QUERY_USER_FOR_DATE(PROMPT: STRING) return TIME is
         use INT_IO;
         MONTH: MONTH_NUMBER;
         DAY: DAY_NUMBER;
         YEAR: YEAR_NUMBER;
         DATE: TIME;
      begin
         loop
            begin
               NEW_LINE;
               PUT("Enter Date of " & PROMPT); NEW_LINE;
               PUT("Month: "); GET(MONTH);
               PUT("Day: "); GET(DAY);
               PUT("Year: "); GET(YEAR);
               DATE := TIME_OF(YEAR, MONTH, DAY);
               exit;
            exception
               when DATA_ERROR | CONSTRAINT_ERROR | TIME_ERROR =>
                  PUT("Invalid DATE"); NEW_LINE;
            end;
         end loop;
         return DATE;
      end QUERY_USER_FOR_DATE;
```

```
procedure SORT_FILE(FILE: FILE_TYPE) is
      EDGE: INDEX_TYPE := 1;
      MAX: INDEX_TYPE;
      EDGE_ITEM: ITEM_TYPE;
      MAX_ITEM: ITEM_TYPE;
      CURRENT_ITEM: ITEM_TYPE;
   begin
      loop
         SET_INDEX(FILE, EDGE);
         exit when END_OF_FILE(FILE);
         READ(FILE, EDGE_ITEM);
         MAX := EDGE;
         MAX_ITEM := EDGE_ITEM;
         while not END_OF_FILE(FILE) loop
            READ(FILE, CURRENT_ITEM);
            if CURRENT_ITEM > MAX_ITEM then
               MAX := INDEX(FILE)-1;
               MAX_ITEM := CURRENT_ITEM;
            end if;
         end loop;
         if MAX /= EDGE then
            WRITE(FILE, MAX_ITEM, EDGE);
            WRITE(FILE, EDGE_ITEM, MAX);
         end if;
         EDGE := EDGE+1;
      end loop;
      SET_INDEX(FILE, 1);
   end SORT_FILE;

procedure SUMMARIZE_FILE(FROM_FILE: FROM_FILE_TYPE; TO_FILE: TO_FILE_TYPE) is
      COUNT: INTEGER := 1;
      CURRENT: FROM_ITEM_TYPE;
      PRIOR: FROM_ITEM_TYPE;
   begin
      if not END_OF_FILE(FROM_FILE) then
         READ(FROM_FILE, PRIOR);
         while not END_OF_FILE(FROM_FILE) loop
            READ(FROM_FILE, CURRENT);
            if CURRENT /= PRIOR then
               WRITE(TO_FILE, BUILD(COUNT, PRIOR));
               COUNT := 1;
               PRIOR := CURRENT;
            else
               COUNT := COUNT+1;
            end if;
         end loop;
         WRITE(TO_FILE, BUILD(COUNT, CURRENT));
         RESET(TO_FILE);
      end if;
   end SUMMARIZE_FILE;
```

```
package body CHAINED_IO is
   procedure CREATE(FILE: in out FILE_TYPE; MODE: FILE_MODE; NAME: STRING) is
      begin
         UNDERLYING_IO.CREATE(FILE, MODE, NAME);
      end CREATE;
   procedure OPEN(FILE: in out FILE_TYPE; MODE: FILE_MODE; NAME: STRING) is
      begin UNDERLYING_IO.OPEN(FILE, MODE, NAME); end OPEN;
   procedure CLOSE(FILE: in out FILE_TYPE) is
      begin UNDERLYING_IO.CLOSE(FILE); end CLOSE;
   procedure READ(FILE: FILE_TYPE; ITEM: out ELEMENT_TYPE;
                  FROM: INDEX_TYPE) is
      begin
         UNDERLYING_IO.READ(FILE, ITEM, UNDERLYING_IO.POSITIVE_COUNT(FROM));
      end READ;
   procedure WRITE(FILE: FILE_TYPE; ITEM: ELEMENT_TYPE; TO: INDEX_TYPE) is
      begin
         UNDERLYING_IO.WRITE(FILE, ITEM, UNDERLYING_IO.POSITIVE_COUNT(TO));
      end WRITE;
   function END_OF_FILE(FILE: FILE_TYPE) return BOOLEAN is
      begin return UNDERLYING_IO.END_OF_FILE(FILE); end END_OF_FILE;
   function FREE_INDEX(FILE: FILE_TYPE) return INDEX_TYPE is
      begin return 1 + UNDERLYING_IO.SIZE(FILE); end FREE_INDEX;
   function NULL_INDEX(INDEX: INDEX_TYPE) return BOOLEAN is
      begin return INDEX = 0; end NULL_INDEX;
   function NULL_INDEX return INDEX_TYPE is
      begin return 0; end NULL_INDEX;
end CHAINED_IO;
```

```
   package body KEYED_IO is
      procedure CREATE(FILE: in out FILE_TYPE; MODE: FILE_MODE; NAME: STRING) is
         begin
            UNDERLYING_IO.CREATE(FILE, MODE, NAME);
         end CREATE;
      procedure OPEN(FILE: in out FILE_TYPE; MODE: FILE_MODE; NAME: STRING) is
         begin UNDERLYING_IO.OPEN(FILE, MODE, NAME); end OPEN;
      procedure CLOSE(FILE: in out FILE_TYPE) is
         begin UNDERLYING_IO.CLOSE(FILE); end CLOSE;
      procedure SET_KEY(FILE: FILE_TYPE; KEY: KEY_TYPE) is
            COMBINED: COMBINED_TYPE;
            I: UNDERLYING_IO.POSITIVE_COUNT;
         begin
            UNDERLYING_IO.RESET(FILE);
            while not UNDERLYING_IO.END_OF_FILE(FILE) loop
               UNDERLYING_IO.READ(FILE, COMBINED);
               if COMBINED.KEY = KEY then
                  I := UNDERLYING_IO.INDEX(FILE)-1;
                  UNDERLYING_IO.SET_INDEX(FILE, I);
                  exit;
               end if;
            end loop;
         end SET_KEY;
      procedure READ(FILE: FILE_TYPE; ITEM: out ELEMENT_TYPE; FROM: KEY_TYPE) is
            COMBINED: COMBINED_TYPE;
         begin
            SET_KEY(FILE, FROM);
            UNDERLYING_IO.READ(FILE, COMBINED);
            ITEM := COMBINED.ELEMENT;
         end READ;
      procedure READ(FILE: FILE_TYPE; ITEM: out ELEMENT_TYPE) is
            COMBINED: COMBINED_TYPE;
         begin
            UNDERLYING_IO.READ(FILE, COMBINED);
            ITEM := COMBINED.DATA;
         end READ;
      procedure WRITE(FILE: FILE_TYPE; ITEM: ELEMENT_TYPE; TO: KEY_TYPE) is
         begin
            SET_KEY(FILE, TO);
            UNDERLYING_IO.WRITE(FILE, COMBINED_TYPE'(TO, ITEM));
         end WRITE;
      function END_OF_FILE(FILE: FILE_TYPE) return BOOLEAN is
         begin return UNDERLYING_IO.END_OF_FILE(FILE); end END_OF_FILE;
   end KEYED_IO;
end FUNCTIONS;
```

References

[Ada 83] "Military Standard Ada Programming Language", U.S. Department of Defense, ANSI/MIL-STD-1815A, January 1983.

[Balzer 81] R. Balzer, "Transformational Implementation: An Example", IEEE TSE V7 #1, January 1981.

[Barstow 77] D.R. Barstow, "Automatic Construction of Algorithms and Data Structures Using A Knowledge Base of Programming Rules", (PhD Thesis) Stanford AIM-308, November 1977.

[Barstow 82] D.R. Barstow, R.D. Duffey, S. Smoliar and S. Vestal, "An Automatic Programming System to Support an Experimental Science", Sixth International Conference on Software Engineering, September 1982.

[Bassett 84] P. Bassett, "Design Principles for Software Manufacturing Tools", Proceedings ACM-84, October 1984.

[Boyle 84] J.M. Boyle, "Program Reusability through Program Transformation", IEEE TSE V10 #5, September 1984.

[Brotsky 84] D. Brotsky, "An Algorithm for Parsing Flow Graphs", (MS Thesis) MIT/AI/TR-704, March 1984.

[Budinsky 85] F. Budinsky, R. Holt and S. Zoky, "SRE - A Syntax Recognizing Editor", Software Practice and Experience, 1985.

[Chapman 82] D. Chapman, "A Program Testing Assistant", CACM V25 #9, September 1982.

[Cheatham 84] T.E. Cheatham, "Reusability Through Program Transformations", IEEE TSE V10 #5, September 1984.

[Cheng 84] T.T. Cheng, E.D. Lock and N.S. Prywes, "Use of Very High Level Languages and Program Generation by Management Professionals", IEEE TSE V10 #5, September 1984.

[Cyphers 82] D.S. Cyphers, "Automated Program Explanation", MIT/AI/WP-237, August 1982.

[Dolotta 76] T.A. Dolotta and J.R. Mashey, "An Introduction to the Programmer's Workbench", Second International Conference on Software Engineering, October 1976.

[Donzeau-Gouge 75] V. Donzeau-Gouge *et. al.*, "A Structure-Oriented Program Editor; A First Step Towards Computer Assisted Programming", Proceedings International Computing Symposium, Antibes France, 1975.

[Duffey 80] R.D. Duffey II, "Formalizing the Expertise of the Assembler Language Programmer", MIT/AI/WP-203, September 1980.

[Faust 81] G.G. Faust, "Semiautomatic Translation of Cobol into Hibol", (MS Thesis) MIT/LCS/TR-256, March 1981.

[Fickas 83] S.F. Fickas, "Automating the Transformational Development of Software", (PhD Thesis) USC Information Sciences Institute, ISI/RR-83-108, March 1983.

[Focus 84] Focus General Information Guide, Information Builders Inc., New York, 1985.

[Frank 80] C. Frank, "A Step Towards Automatic Documentation", MIT/AI/WP-213, December 1980.

[Freudenberger 83] S.M. Freudenberger, J.T. Schwartz and M. Sharir, "Experience with the SETL Optimizer", ACM TOPLAS V1 #1, January 1983.

[Green 81] C. Green *et. al.*, "Research on Knowledge-Based Programming and Algorithm Design -- 1981", Kestrel Institute, Palo Alto CA, 1981.

[Green & Rich 83] C. Green, D. Luckam, R. Balzer, T. Cheatham and C. Rich, "Report on a Knowledge-Based Software Assistant", Rome Air Development Center, RADC-TR-83-195, August 1983.

[Harandi 83] M. Harandi, "A Knowledge-Based Programming Support Tool", Proceedings IEEE Trends and Applications Conference, May 1983.

[Kant 79] E. Kant, "Efficiency Considerations in Program Synthesis: A Knowledge-Based Approach", (PhD Thesis) Stanford AIM-331, September 1979.

[Laubsch 81] J. Laubsch and M. Eisenstadt, "Domain Specific Debugging Aids for Novice Programmers", proceedings IJCAI-81, August 1981.

[Lisp 84] Lisp Machine documentation (release 4), Symbolics, Cambridge MA, 1984.

[Medina-Mora 81] R. Medina-Mora and P. Feiler, "An Incremental Programming Environment", IEEE TSE V7 #5, September 1981.

[Mitchell 85] T.M. Mitchell, L.I. Steinberg and J.S. Shulman, "A Knowledge-Based Approach to Design", Rutgers LCSR-TR-65, January 1985.

[Neighbors 84] J.M. Neighbors, "The Draco Approach to Constructing Software from Reusable Components", IEEE TSE V10 #5, September 1984.

[Pitman 83] K.M. Pitman, "Interfacing to the Programmer's Apprentice", MIT/AI/WP-244, February 1983.

[Reps 83] T. Reps, T. Teitelbaum and A. Demers, "Incremental Context-Dependent Analysis for Language-Based Editors", ACM TOPLAS V5 #3, July 1983.

[Reuse 84] Special Issue on Software Reusability, IEEE TSE V10 #5, September 1984.

[Rich 80] C. Rich, "Inspection Methods in Programming", (PhD thesis) MIT/AI/TR-604, June 1981.

[Rich 81] C. Rich, "A Formal Representation for Plans in the Programmer's Apprentice", proceedings IJCAI-81, August 1981.

[Rich 82] C. Rich, "Knowledge Representation Languages and Predicate Calculus: How to Have Your Cake and Eat It Too", proceedings AAAI-82, August 1982.

[Rich 85] C. Rich, "The Layered Architecture of a System for Reasoning about Programs", proceedings IJCAI-85, August 1985.

[Rich & Shrobe 76] C. Rich and H.E. Shrobe, "Initial Report On A Lisp Programmer's Apprentice", (MS Thesis) MIT/AI/TR-354, December 1976.

[Rich & Shrobe 78] C. Rich and H.E. Shrobe, "Initial Report on A Lisp Programmer's Apprentice", IEEE TSE V4 #5, November 1978.

[Rich & Waters 79] C. Rich, H.E. Shrobe and R.C. Waters, "Computer Aided Evolutionary Design for Software Engineering", MIT/AIM-506, January 1979.

[Rich & Waters 81] C. Rich and R.C. Waters, "Abstraction, Inspection and Debugging in Programming", MIT/AIM-634, June 1981.

[Rich & Waters 82] C. Rich and R.C. Waters, "The Disciplined Use of Simplifying Assumptions", Proceedings of ACM SIGSOFT Second Software Engineering Symposium: Workshop on Rapid Prototyping, ACM SIGSOFT Software Engineering Notes V7 #5, December 1982.

[Rich & Waters 83] C. Rich and R.C. Waters, "Formalizing Reusable Software Components", Proceedings ITT Workshop on Reusability in Programming, September 1983.

[Ruth 81] G. Ruth, S. Alter and W. Martin, "A Very High Level Language for Business Data Processing", MIT/LCS/TR-254, 1981.

[Schwartz 75] J.T. Schwartz, "On Programming", An Interim Report on the SETL Project, Courant Institute of Mathematical Sciences, New York University, June 1975.

[Shapiro 81] D. Shapiro, "Sniffer: a System that Understands Bugs", (MS Thesis) MIT/AIM-638, June 1981.

[Shapiro 83] D. Shapiro and B. McCune, "The Intelligent Program Editor", Proceedings IEEE Trends and Applications Conference, May 1983.

[Shrobe 79] H.E. Shrobe, "Dependency Directed Reasoning for Complex Program Understanding", (PhD Thesis) MIT/AI/TR-503, April 1979.

[Soloway 83] E. Soloway *et. al.*, "MENO-II: An Intelligent Programming Tutor", Journal of Computer-Based Instruction V10 #1, Summer 1983.

[Soloway 84] E. Soloway and K. Ehrlich, "Empirical Studies of Programming Knowledge", IEEE TSE V10 #5, September 1984.

[Stallman 81] R. Stallman, "Emacs the Extensible, Customizable, Self-Documenting Display Editor", Proceedings of ACM SIGPLAN-SIGOA Symposium on Text Manipulation, ACM SIGPLAN Notices V16 #6, June 1981.

[Sterpe 85] P.J. Sterpe, "TEMPEST -- A Template Editor for Structured Text", (MS Thesis) MIT/AI/TR-843, May 1985.

[Sussman 79] G.J. Sussman, J. Holloway and T. Knight, "Computer Aided Evolutionary Design for Digital Integrated Systems", MIT/AIM-526, May 1979.

[Teitelbaum 81] T. Teitelbaum and T. Reps, "The Cornell Program Synthesizer: A Syntax-Directed Programming Environment", CACM V24 #9, September 1981.

[Teichroew 77] D.E. Teichroew and E.A. Hershey III, "PSL/PSA: A Computer-Aided Technique for Structured Documentation and Analysis of Information Processing Systems", IEEE TSE V3 #1, January 1977.

[Waters 76] R.C. Waters, "A System for Understanding Mathematical FORTRAN Programs", MIT/AIM-368, May 1976.

[Waters 78] R.C. Waters, "Automatic Analysis of the Logical Structure of Programs", (PhD Thesis) MIT/AI/TR-492, December 1978.

[Waters 79] R.C. Waters, "A Method for Analyzing Loop Programs", IEEE TSE V5 #3, May 1979.

[Waters 82a] R.C. Waters, "The Programmer's Apprentice: Knowledge Based Program Editing", IEEE TSE V8 #1, January 1982.

[Waters 82b] R.C. Waters, "Program Editors Should Not Abandon Text Oriented Commands", ACM SIGPLAN Notices V17 #7, July 1982.

[Waters 83a] R.C. Waters, "LetS: An Expressional Loop Notation", MIT/AIM-680a, February 1983.

[Waters 83b] R.C. Waters, "User Format Control in a Lisp Prettyprinter", ACM TOPLAS V5 #4, October 1983.

[Waters 84a] R.C. Waters, "Expressional Loops", Proceedings ACM SIGACT-SIGPLAN Symposium on the Principles of Programming Languages, January 1984.

[Waters 84b] R.C. Waters, "PP: A Lisp Pretty Printing System", MIT/AIM-816, December 1984.

[Wile 82] D.S. Wile, "Program Developments: Formal Explanations of Implementations", USC Information Sciences Institute, ISI/RR-82-99, August 1982.

[Zelinka 83] L.M. Zelinka, "An Empirical Study of Program Modification Histories", MIT/AI/WP-240, February 1983.

CS-TR Scanning Project
Document Control Form

Date : 2 / 6 / 96

Report # AI-TR-753

Each of the following should be identified by a checkmark:
Originating Department:

☒ Artificial Intellegence Laboratory (AI)
☐ Laboratory for Computer Science (LCS)

Document Type:

☒ Technical Report (TR) ☐ Technical Memo (TM)
☐ Other: ____________

Document Information

Number of pages: 242 (248-IMAGES)
Not to include DOD forms, printer intstructions, etc... original pages only.

Originals are:
☒ Single-sided or
☐ Double-sided

Intended to be printed as :
☐ Single-sided or
☒ Double-sided

Print type:
☐ Typewriter ☐ Offset Press ☐ Laser Print
☐ InkJet Printer ☐ Unknown ☒ Other: ____________

Check each if included with document:

☒ DOD Form (2) ☐ Funding Agent Form ☐ Cover Page
☐ Spine ☐ Printers Notes ☐ Photo negatives
☐ Other: ____________

Page Data:

Blank Pages (by page number): 74, 136, 154, 196, 232

Photographs/Tonal Material (by page number): ____________

Other (note description/page number):

Description : Page Number:

(A) IMAGE MAP (1-242) UN#'ED TITLE, BLANK, DEDICATION, ACKNOWLEDGEMENT, CONTENTS, BLANK PAGES, 1-236
(243-248) SCANCONTROL, DOD (2), TRGT'S (3)
(B) CUTOFF TEXT FIG'S ON PAGES 2, 7, 10, 12-13, 155, 164, 168, 169, 173, 176

Scanning Agent Signoff:
Date Received: 2 / 6 / 96 Date Scanned: 2 / 7 / 96 Date Returned: 2 / 15 / 96

Scanning Agent Signature: Michael W. Cook

Rev 9/94 DS/LCS Document Control Form cstrform.vsd

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE		READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER AI-TR-753	2. GOVT ACCESSION NO.	3. RECIPIENT'S CATALOG NUMBER
4. TITLE (and Subtitle) KBEmacs: A Step Toward the Programmer's Apprentice		5. TYPE OF REPORT & PERIOD COVERED Technical Report
		6. PERFORMING ORG. REPORT NUMBER
7. AUTHOR(s) Richard C. Waters		8. CONTRACT OR GRANT NUMBER(s) N00014-80-C-0505 MCS-7912179 MCS-8117633
9. PERFORMING ORGANIZATION NAME AND ADDRESS Artificial Intelligence Laboratory 545 Technology Square Cambridge, MA 02139		10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS
11. CONTROLLING OFFICE NAME AND ADDRESS Advanced Research Projects Agency 1400 Wilson Blvd. Arlington, VA 22209		12. REPORT DATE May,1985
		13. NUMBER OF PAGES 236
14. MONITORING AGENCY NAME & ADDRESS(if different from Controlling Office) Office of Naval Research Information Systems Arlington, VA 22217		15. SECURITY CLASS. (of this report) Unclassified
		15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this Report)

Distribution is unlimited

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

18. SUPPLEMENTARY NOTES

None

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

Computer Aided Design
Program Editing
Programming Enviornment
Reuseable Software Components
Programming Apprentice

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

The Knowledge-Based Editor in Emacs (KBEmacs) is the current demonstration system as part of the Programmers Apprentice project. KBEmacs is capable of acting as a semi-expert assistant to a person who is writing a program--taking over some parts of the programming task. Using KBEmacs,it is possible to construct a program issuing a series of high level commands. This series of commands can be as much as an order of magnitude shorter than the program it describes.

20. KBEmacs is capable of operating on programs of realistic size and coplexity. Although KBEmacs is neither fast enough nor robust enough to be considered a true prototype, both of these problems could be overcome if the system were to be reimplemented.